

Mira Mezini (Ed.)

LNCS 6813

ECOOP 2011 – Object-Oriented Programming

25th European Conference
Lancaster, UK, July 2011
Proceedings

 Springer

Commenced Publication in 1973

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison

Lancaster University, UK

Takeo Kanade

Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler

University of Surrey, Guildford, UK

Jon M. Kleinberg

Cornell University, Ithaca, NY, USA

Alfred Kobsa

University of California, Irvine, CA, USA

Friedemann Mattern

ETH Zurich, Switzerland

John C. Mitchell

Stanford University, CA, USA

Moni Naor

Weizmann Institute of Science, Rehovot, Israel

Oscar Nierstrasz

University of Bern, Switzerland

C. Pandu Rangan

Indian Institute of Technology, Madras, India

Bernhard Steffen

TU Dortmund University, Germany

Madhu Sudan

Microsoft Research, Cambridge, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max Planck Institute for Informatics, Saarbruecken, Germany

Mira Mezini (Ed.)

ECOOP 2011 – Object-Oriented Programming

25th European Conference
Lancaster, UK, July 25-29, 2011
Proceedings

Volume Editor

Mira Mezini
Darmstadt University of Technology
Department of Computer Science
Hochschulstraße 10, 64289 Darmstadt, Germany
E-mail: mezini@informatik.tu-darmstadt.de

ISSN 0302-9743
ISBN 978-3-642-22654-0
DOI 10.1007/978-3-642-22655-7
Springer Heidelberg Dordrecht London New York

e-ISSN 1611-3349
e-ISBN 978-3-642-22655-7

Library of Congress Control Number: 2011932356

CR Subject Classification (1998): D.1, D.2, D.3, F.3, C.2, H.4, J.1

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2011

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

It is an honor and a pleasure to write this preface to the proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011) held in Lancaster, UK.

ECOOP is the premier European conference on object-oriented programming and related software development paradigms. The ECOOP research track brings together leading researchers and practitioners working in the fields of software engineering, programming languages, and software systems. This year, it consisted of 26 papers included in these proceedings, which the Program Committee carefully selected from a total of 100 submitted papers.

Each paper received at least three reviews from a Program Committee consisting of 30 internationally reputed researchers. A long and intensive virtual discussion was conducted after the first versions of the reviews were available and prior to an author response period, during which the authors had the opportunity to respond to reviews in depth and without a-priori length limit to clear up misunderstandings or to answer questions specifically posed by reviewers. The response period was followed by another series of online discussions rounded off by a Program Committee meeting late February in Darmstadt, where final decisions were made. The Champion pattern was used: for a paper to be accepted at least one manifest proponent is required. Program Committee members were allowed to submit a paper, but these were subjected to a higher level of scrutiny.

The selected program reflected how ECOOP brings together different communities around the theme of software development paradigms and tools. There were papers that presented very interesting findings resulting from comprehensive empirical studies. Another group of papers were dedicated to approaches, techniques, and tools for assisting software developers in using APIs, or evolving software systems. Aliasing and ownership were also prominent topics in the program as well as other traditionally represented topics at ECOOP editions such as types, memory and runtime optimization techniques, modeling, refactoring, and exception handling. A very interesting group of papers concerned with various aspects of modularity rounded up a very strong and diverse program.

I would like to thank the many authors who contributed by submitting a paper, in spite of the high requirements. I would like to express my profound gratitude and my highest respect to the members of the Program Committee that put an incredible amount of effort to enable a critical but fair selection and to provide thorough and constructive feedback to the authors. Many thanks also to all co-reviewers who generously shared their expertise during the evaluation process. Richard van de Stadt of Borbala Online Conference Services provided superb support, going far beyond the call of duty. Gudrun Harris, Eric Boddien and Martin Monperrus assisted with all organizational aspects of the Program Committee meeting, as did many other members of the Software Technology team of Darmstadt University of Technology, Germany.

Continuing the tradition of previous conferences, ECOOP 2011 was more than just the research track: it included keynote speeches by this year's Dahl-Nygaard Prize winners, Craig Chambers and Atsushi Igarashi, a keynote speech by Erik Meijer, a banquet speech by the Turing Award winner, Alan Key, an excellent summer school program, several workshops, demonstrations, social events and more.

May 2011

Mira Mezini

Organization

ECOOP 2011 was organized by the School of Computing and Communications, Science and Technology Department at Lancaster University, under the auspices of AITO (Association Internationale pour les Technologies Objets), and in cooperation with ACM SIGPLAN and ACM SIGSOFT.



Conference Chair

Awais Rashid Lancaster University, UK

Program Chair

Mira Mezini Darmstadt University of Technology, Germany

Organizing Chairs

Phil Greenwood Lancaster University, UK
Ruzanna Chitchyan Lancaster University, UK

Social Program and Conference Management

Yvonne Rigby Lancaster University, UK

Workshop Chairs

Sophia Drossopoulou Imperial College London, UK
Eric Eide University of Utah, USA

Summer School Chairs

James Noble	Victoria University of Wellington, New Zealand
Jan Vitek	Purdue University, USA

Publicity Chairs

Joao Araujo	Universidade Nova de Lisboa, Portugal
Joost Noppen	University of East Anglia, UK
Alberto Sardinha	Instituto Superior Técnico, Portugal

Poster and Demos Chairs

Hidehiko Masuhara	University of Tokyo, Japan
Ana Moreira	Universidade Nova de Lisboa, Portugal

Research Projects Symposium Chairs

Steffen Zschaler	King's College London, UK
Geir Horn	Sintef, Norway

Web Chair

Raffi Khatchadourian	Ohio State University, USA
----------------------	----------------------------

Gold Sponsors



Silver Sponsors

Microsoft
Research **IBM Research** **Google**

Program Committee

Jonathan Aldrich	Carnegie Mellon University, USA
Gustavo Alonso	ETH Zürich, Switzerland
Sven Apel	University of Passau, Germany
Gavin Bierman	Microsoft Research, UK
Andrew P. Black	Portland State University, USA
Gilad Bracha	Ministry of Truth, USA
Arie van Deursen	Delft University of Technology, The Netherlands
Theo D'Hondt	Vrije Universiteit Brussel, Belgium
Sophia Drossopoulou	Imperial College London, UK
Erik Ernst	Aarhus University, Denmark
Patrick Eugster	Purdue University, USA
Dick Gabriel	IBM Research, USA
David Grove	IBM Research, USA
Robert Hirschfeld	Hasso-Plattner Institute, Postdam, Germany
Jean-Marc Jézéquel	Rennes 1 and INRIA, France
Doug Lea	State University of New York at Oswego, USA
Crista Lopes	University of California, Irvine, USA
Todd Millstein	University of California, Los Angeles, USA
James Nobles	Victoria University of Wellington, New Zealand
Klaus Ostermann	University of Marburg, Germany
Arnd Poetzsch-Heffter	University of Kaiserslautern, Germany
Awais Rachid	Lancaster University, UK
Frank Tip	IBM T.J. Watson Research Center, USA
Laurie Tratt	Middlesex University, UK
Adam Welc	Intel Labs, USA
Andreas Zeller	Saarland University, Saarbrücken, Germany
Matthias Zenger	Google
Thomas Zimmermann	Microsoft Research
Elena Zucca	DISI, University of Genoa, Italy

Referees

George Allan	Bruno Cabral
Davide Ancona	Nicholas Cameron
Kelly Androutsopoulos	Walter Cazzola
Malte Appeltauer	Susanne Cech Previtali
Engineer Bainomugisha	Maura Cerioli
Stephanie Balzer	Mickael Clavreul
Lorenzo Bettini	Pascal Costanza
Robert Bocchino	Bruno De Fraine
Eric Bodden	Christoph Feller
Cedric Bouhours	Jean-Marie Gaillourdet
Johann Bourcier	Kathrin Geilmann

Felix Geller
Michaela Greiler
Michael Haupt
Florian Heidenreich
Michael Hicks
Ciera Jaspan
Niels Joncheere
Andy Kellens
Andrew Kennedy
Charles E. Killian
Matthias Kleine
Robert Krahn
Neelakantan R. Krishnaswami
Martin Kuhlemann
Ilham Kurnia
Christian Kästner
Giovanni Lagorio
Gary Leavens
Jens Lincke
Andoni Lombide Carreton

Patrick Michel
Filipe Militão
Matthew Parkinson
Gilles Perrouin
Michael Perscheid
Hesam Samimi
Ina Schaefer
Jan Schäfer
Stefan Sobernig
Bastian Steinert
Sven Stork
Joshua Sunshine
Gerson Sunyé
Thomas Thüm
Stijn Timbermont
Salvador Trujillo
Veronica Uquillas-Gomez
Tom Van Cutsem
Yannick Welsch
Roger Wolff

Table of Contents

Keynote 1

A Co-relational Model of Data for Large Shared Data Banks	1
<i>Erik Meijer</i>	

Empirical Studies

An Empirical Study of Object Protocols in the Wild	2
<i>Nels E. Beckman, Duri Kim, and Jonathan Aldrich</i>	
The Beauty and the Beast: Separating Design from Algorithm	27
<i>Dmitrijs Zaparanuks and Matthias Hauswirth</i>	
The Eval That Men Do: A Large-Scale Study of the Use of Eval in JavaScript Applications	52
<i>Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek</i>	

Mining, Understanding, Recommending

Using Structure-Based Recommendations to Facilitate Discoverability in APIs	79
<i>Ekwa Duala-Ekoko and Martin P. Robillard</i>	
Mining Evolution of Object Usage	105
<i>Yana Momchilova Mileva, Andrzej Wasylkowski, and Andreas Zeller</i>	
Improving the Tokenisation of Identifier Names	130
<i>Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp</i>	

Modularity

Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity	155
<i>Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel</i>	
Worlds: Controlling the Scope of Side Effects	179
<i>Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay</i>	
Can We Avoid High Coupling?	204
<i>Craig Taube-Schock, Robert J. Walker, and Ian H. Witten</i>	

Keynote 2

Expressiveness, Simplicity, and Users	229
<i>Craig Chambers</i>	

Modelling and Refactoring

CDDiff: Semantic Differencing for Class Diagrams	230
<i>Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe</i>	
A Refactoring Constraint Language and Its Application to Eiffel	255
<i>Friedrich Steimann, Christian Kollee, and Jens von Pilgrim</i>	
Modal Object Diagrams	281
<i>Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe</i>	

Aliasing and Ownership

Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks	306
<i>Robert L. Bocchino Jr. and Vikram S. Adve</i>	
Tunable Static Inference for Generic Universe Types	333
<i>Werner Dietl, Michael D. Ernst, and Peter Müller</i>	
Verifying Multi-object Invariants with Relationships	358
<i>Stephanie Balzer and Thomas R. Gross</i>	

Memory Optimizations

Patterns of Memory Inefficiency	383
<i>Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O’Sullivan, Trevor Parsons, and John Murphy</i>	
Reuse, Recycle to De-bloat Software	408
<i>Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta</i>	

Keynote 3

A Featherweight Approach to FOOL	433
<i>Atsushi Igarashi</i>	

Types

Related Types	434
<i>Johnni Winther and Michael I. Schwartzbach</i>	

Gradual Typestate	459
<i>Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich</i>	
Maintaining Database Integrity with Refinement Types	484
<i>Ioannis G. Baltopoulos, Johannes Borgström, and Andrew D. Gordon</i>	

Runtime and Memory Optimizations

Frequency Estimation of Virtual Call Targets for Object-Oriented Programs	510
<i>Cheng Zhang, Hao Xu, Sai Zhang, Jianjun Zhao, and Yuting Chen</i>	
Counting Messages as a Proxy for Average Execution Time in Pharo ...	533
<i>Alexandre Bergel</i>	
Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging	558
<i>Guillaume Pothier and Éric Tanter</i>	

Exceptions

Interprocedural Exception Analysis for C++	583
<i>Prakash Prabhu, Naoto Maeda, Gogul Balakrishnan, Franjo Ivančić, and Aarti Gupta</i>	
Detecting and Escaping Infinite Loops with Jolt	609
<i>Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard</i>	
Atomic Boxes: Coordinated Exception Handling with Transactional Memory	634
<i>Derin Harmanci, Vincent Gramoli, and Pascal Felber</i>	
Author Index	659

A Co-relational Model of Data for Large Shared Data Banks

Erik Meijer

Microsoft Research

Abstract. For the past decade, I have been on a quest to democratize developing data-intensive distributed applications. My secret weapon to slay the complexity dragon has been category theory and monads, but in particular the concept of duality. As it turns out, the data domain is an extremely rich source of all kinds of interesting dualities. These dualities are not just theoretical curiosities, but actually solve many practical problems and help to uncover deep similarities between concepts that at first look totally unrelated. In this talk I will illustrate several of the dualities I have encountered during my journey, and show how this resulted in a novel “A co-Relational Model of Data for Large Shared Data Banks”.

An Empirical Study of Object Protocols in the Wild

Nels E. Beckman, Duri Kim, and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, USA
{nbeckman,aldrich}@cs.cmu.edu, duri.kim@alumni.cmu.edu

Abstract. An active area of research in computer science is the prevention of violations of object protocols, i.e., restrictions on temporal orderings of method calls on an object. However, little is understood about object protocols in practice. This paper describes an empirical study of object protocols in some popular open-source Java programs. In our study, we have attempted to determine how often object protocols are defined, and how often they are used, while also developing a taxonomy of similar protocols. In the open-source projects in our study, comprising almost two million lines of code, approximately 7.2% of all types defined protocols, while 13% of classes were clients of types defining protocols. (For comparison, 2.5% of the types in the Java library define type parameters using Java Generics.) This suggests that protocol checking tools are widely applicable.

1 Introduction

Object protocols are rules dictating the ordering of method calls on objects of a particular class. We say that a type defines an *object protocol* if its concrete state can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition (a definition we will expand in Section 2.1). The classic example of an object protocol, often cited in research literature, is that of a file class. Instances of this file class can only have their **read** methods called while the file is open. Once the file is closed with the **close** method, subsequent calls to the **read** method will result in run-time exceptions or undefined behavior. Most popular languages do not give object protocols first-class status, and therefore cannot ensure their correct use statically.

Static and dynamic checking of object protocols is an extremely active area of research in the software engineering and programming languages communities. There have been protocol checkers based on software model checking [3, 12]. There have been type systems and flow analyses for checking object protocols [23, 8, 19, 5]. (Type-based checkers are so common that these properties are often referred to as “typestate” properties.) There have been checkers that focus on the narrower problem of object initialization [10, 22], and checkers that focus on the wider issues of framework conformance [16, 11]. There have even been dynamic checkers [14, 6], and checkers that focus on concurrent applications [4, 17].

While many of these approaches are quite powerful and their designs elegant, we argue that very little is known about how protocols are used in practice. Do they occur often or are they rarely defined? Are they used by many other classes? Are the protocols themselves simple, or complex? These are the kinds of questions we have attempted to answer with this study.

In this paper, we present an empirical study on object protocols in open source Java software. We took several popular open-source projects and the Java standard library, ran a suite of automated analyses that attempted to find evidence of object protocols, and manually investigated the results.

This work contains several contributions. As part of our investigation, we discovered that object protocol definition is relatively common (in about 7% of all types) and protocol use even more so (by about 13% of all classes). We discovered seven behavioral categories of object protocols that account for 98% of all the protocols we discovered. Compared to existing protocol studies which looked at large volumes of code [26, 27, 2], ours is the first to attempt to examine characteristics of the protocols themselves, for example frequency of definition and categories of protocols.

The paper proceeds in the following manner: Section 2 discusses the design of our experiment. This includes important definitions, description of our automated analyses, the data that we gathered and the motivation underlying our approach. Section 2.4 describes the threats to the validity of our experiment. Section 3 presents the data that we gathered during our study, and Section 4 discusses that data and its implications for other researchers.

2 Methodology

Our study proceeded in the following manner: We created a static analysis to detect patterns in source code that we believe are indicative of object protocols. Then, we ran the static analysis on popular open-source Java projects and the Java standard library. Next, we manually investigated the reports issued by the static analysis, marking each as evidence for a protocol or not. During this process, data about the location, classes involved, their super-types, and more was gathered. We also created categories of similar protocols based on our observations. Finally, we used the information about which types define protocols in order to run another automated analysis which gathered information about the usage of those protocols. An earlier version of this study was conceived by one author, Duri Kim, and presented in her masters thesis [18].

The first part of this section will discuss our definitions, namely, what are object protocols? Next we walk the reader through the experimental process, including a description of our analyses and the data we gathered. Finally, we describe the Java programs we analyzed and threats to the validity of our study.

2.1 Definitions and Scope

One of the trickiest parts of discussing object protocols is agreeing on exactly what is meant by the term. While many sanctioned interactions between dif-

ferent pieces of code could be described generically using the term protocol, we choose to focus on a definition that is based around abstract state machines. The definition of object protocol stated here sets the scope for our entire experiment. It is the idea on which our analyses and terms like “false negative” will be based.

Definition. A type defines an *object protocol* if the concrete state of objects of that type can be abstracted into a finite number of abstract states of which clients must be aware in order to use that type correctly, and among which object instances will dynamically transition.

This definition contains several key ideas.

client. The states of the protocol must be observable and relevant to clients.

abstract and finite. The states must be abstractions of any internal representation, and there must be a finite number.

runtime transitions. Methods calls on an object instance after construction will cause it to transition between abstract states.

correct use. Failure of clients to obey a protocol can result in run-time exceptions or undefined behavior.

Classic examples of protocols fall under this definition. For example, an instance of the `java.io.FileReader` class can be interpreted as having two abstract states, `Open` and `Closed`. Clients must be aware of which state a given instance of the file is in otherwise they might incorrectly call a method such as `read`, which requires the file to be open, when the file is actually closed. `java.util.Iterator` fits our definition as well. Even though it is an interface and does not have its own concrete state, clients must be aware that the `next` method can only be called when a call to `hasNext` would return true.

Our definition includes initialization protocols; objects that must have certain methods called after construction to put them into a valid, *initialized* state. While these protocols may in fact be quite simple, they fit our definition, and are an important piece of the contract of many types.

We additionally include a degenerate form of protocol known as type qualifiers [13, 9]. In this case, object instances enter an abstract state at construction-time that they can never leave. Like other protocols, depending on the state the object is in, certain method calls may be illegal. We will point out type qualifiers in this study even though they do not strictly fit our definition, as we feel they are quite similar to more standard object protocols and because, like object protocols, current languages do not check them statically.

Our definition specifically excludes protocols in which a type has an infinite number of abstract states. This is meant to exclude types such as `java.util.List` on the basis of methods like `List.remove(int)`. This method throws an exception when the argument is greater than or equal to the size of the list. While `List` could be interpreted as having the abstract states, `LargerThan0`, `LargerThan1`, `LargerThan2`, etc., this does not fall under our definition, and will not be considered a protocol.

Scope of this Study. Our definition of object protocol leaves out other object protocols that some readers may consider to be important. For example, it does

not include multi-object protocols, in which clients must call an ordered sequence of methods on two or more objects. One of the things we will show is that, even when taking a restricted view of object protocols, they are still rather common. By considering a more inclusive definition, we believe one would find that object protocols are even more common.

We have observed that protocol classes frequently are implemented so that they can detect protocol violations. Generally, violations that are detected will cause an exception to be thrown (e.g., `InvalidStateException`). This is relevant to our study because, within the scope of our definition of object protocol, our automated analysis detects the *subset* for which this is true (see Section 2.2).

Other Definitions. Here are some other terms that will be used throughout the remainder of the paper:

Phase 1. In the first phase of the study we examined the nature of protocol definition.

Phase 2. In the second phase of the study we examined protocol use.

Candidate, Candidate Code. A section of code that may represent evidence of an object protocol, as reported by our static analysis.

Protocol Evidence. A candidate that, after manual analysis, is determined to be evidence of an object protocol (a true positive).

Evidence Class. A class that contains protocol evidence.

2.2 Experimental Procedure

Our experiment consisted of several steps where we alternatingly performed analyses, manual and automated, and gathered and processed their results. This section presents the entire process from start to finish. For convenience, this process is illustrated in Figure 1. At each step in the experiment, we will say what data is gathered and why that particular course of action was chosen.

Phase 1: Finding Object Protocols. In the first phase of our experiment, we start with a set of programs in which we would like to find object protocols. The first step is to run ProtocolFinder, a static analysis that will generate a list of code candidates, locations in code that *may* indicate that a class is defining an object protocol.

We had several goals in mind when developing the ProtocolFinder static analysis. For one, we wanted to keep the rate of false negatives as low as possible. In this case, false negatives are protocols that exist in the programs under analysis that are not found. Manual inspection, of course, can have a very low rate of false negatives but is extremely time consuming, particularly considering the amount of code we would like to investigate. We desired an automated analysis. Dynamic analyses for discovering protocols in running programs exist [15, 28]. Unfortunately, such approaches are quite susceptible to false negatives, since appropriate test cases must be found to exercise all of the possible protocols in an application. For the same reasons, a dynamic approach would require examining only programs that were accompanied by sufficient test cases, and thus,

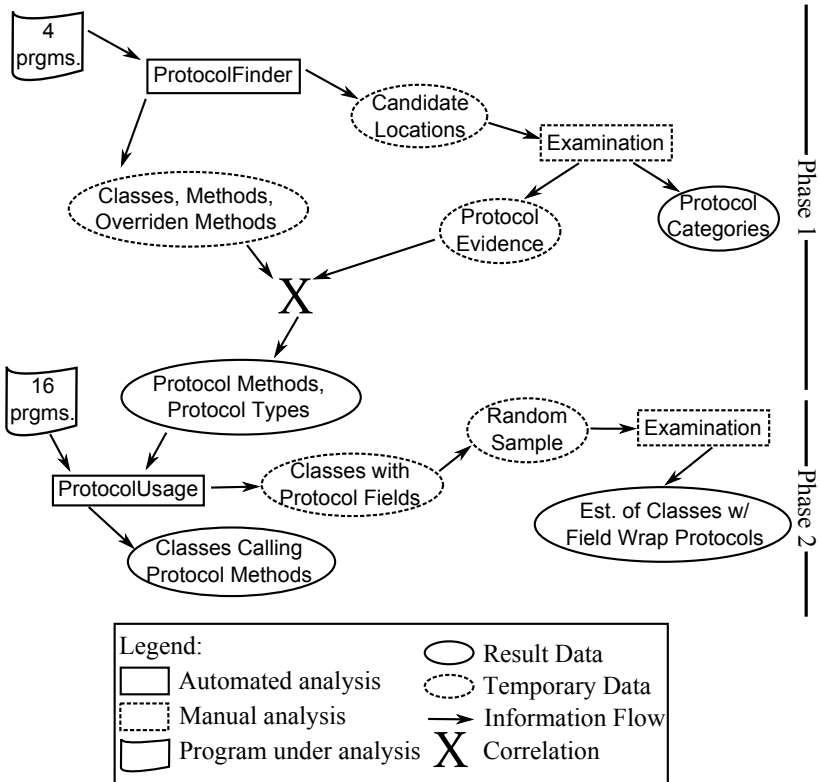


Fig. 1. A schematic explaining the experimental procedure

was ruled out. By comparison, a static analysis can be run on any open-source program. In the end, we decided to develop a conservative static analysis that would eliminate many (although not all) false negatives while reducing manual effort. A subsequent manual examination is used to eliminate false positives.

ProtocolFinder is a static analysis created for this study that attempts to find object protocols by searching for locations in code where protocol violations are detected. Specifically, it looks for locations in code where instance methods throw exceptions as a result of reading instance fields.

The intuition behind the analysis is simple: In our preliminary investigations we noticed that many protocol methods throw exceptions when object protocols are violated. Because our definition of object protocol depends on some abstract state of the method *receiver*, we expect that any exceptions thrown for protocol violation will be thrown in instance methods and as a result of reading an instance field. This pattern has been noted and used as the basis for existing protocol detectors [27, 2]. Like ProtocolUsage, described later, the ProtocolFinder analysis is an Eclipse plugin whose source we have made freely available.¹

¹ <http://code.google.com/p/nolacoaster/>

```

1 // from java.util.concurrent.ArrayBlockingQueue.Itr
2 public void remove() {
3     final ReentrantLock lock = ArrayBlockingQueue.this.lock;
4     lock.lock();
5     try {
6         int i = this.lastRet;
7         if (i == -1)
8             throw new IllegalStateException();
9         lastRet = -1;
10        // ... method continues
11    }
12
13 // from javax.swing.undo.AbstractUndoableEdit
14 public void undo() throws CannotUndoException {
15     if (!canUndo()) {
16         throw new CannotUndoException();
17     }
18     hasBeenDone = false;
19 }
20
21 public boolean canUndo() { return alive && hasBeenDone; }

```

Fig. 2. The ProtocolFinder reports candidate code on lines [8](#) and [16](#). Both are classified as protocol evidence. In the first, the field `lastRet` flows through a local variable `i`. In the second, the field value comes from a getter.

ProtocolFinder. ProtocolFinder is a flow-insensitive static analysis that examines every instance method in a given code base. Upon encountering an ‘if’ block or a conditional expression, the analysis first examines the condition. If the condition expression contains a read of a field of the current receiver (or a call to a “getter” method on the current receiver), the analysis will examine both ‘then’ and ‘else’ branches. (“Getter” methods are methods which more or less immediately return the value of a field.) If either branch of the conditional throws an exception the analysis issues a report indicating that piece of code is a protocol candidate. Both the field read in the condition and the throw statement in the branches can be nested arbitrarily deeply. In order to determine whether an expression in the condition is a field read or getter call on the current receiver, the analysis queries a sub-analysis. This flow-sensitive static analysis has a list of all the methods in the current class determined to be field getters, and can track if a value in an intermediate variable flows from a getter or a field.

The analysis uses a simple procedure to determine which methods are “getter” methods. Any method with a non-void return type where all return statements contain values that flow from field reads are marked as getters.

ProtocolFinder reports protocol candidates in the examples shown in Figures [2](#) and [3](#), all of which are from the Java standard library. In Figure [2](#), reports are issued on lines [8](#) and [16](#). The first example comes from an implementation of the `Iterator` interface. It is noteworthy because the field value flows from the

```

1 // from java.awt.Container
2 public void remove(int index) {
3     synchronized (getTreeLock()) {
4         if(index<0||index>=this.component.size()){
5             throw new ArrayIndexOutOfBoundsException(index);
6         }
7     // ... method continues
8 }

```

Fig. 3. In the `remove` method the ProtocolFinder reports a possible protocol on line 5. Note that the field, `component`, is nested in a sub-expression of the condition (line 4). By manual examination, we have determined that this candidate is *not* evidence for an object protocol.

`lastRet` field to the local variable `i` before the conditional. The second example is noteworthy because the condition involves a call to the getter method `canUndo`, which itself is the result of a combination of fields, `alive` and `hasBeenDone`.

In Figure 3, ProtocolFinder reports a candidate on line 5. This example is noteworthy because the field `read` that occurs on line 4 is nested within a sub-expression of the condition. ProtocolFinder still treats the condition as being dependent on a receiver field.

The output of the ProtocolFinder is thus a list of protocol *candidates*. In the next part of the experiment, we manually inspect each candidate to determine whether or not it is actually evidence of an object protocol. For each report issued, the ProtocolFinder includes the line number and file name of the candidate, the method and class in which the candidate was found, and all methods that are overridden by the method in which the candidate was found. This information helps us find the candidate for the purposes of manual examination, and, in the event that a candidate represents evidence of an actual protocol, will provide us with the data we need to carry out the usage phase of our study.

Manual Examination. After running the ProtocolFinder and gathering a list of protocol candidates, we investigated each candidate by hand.² The primary purpose of this manual investigation was to determine which candidates were actual evidence of object protocols and which were not. This was done by looking at the code location and the surrounding class and trying to understand its behavior. Where possible, documentation was also examined. After understanding the candidate and the conditions under which an exception would be thrown, we consulted our own definition of object protocol in order to determine whether or not the candidate represented protocol evidence.

As an example, consider the code snippets in Figures 2 and 3. Both were returned as candidates by the ProtocolFinder. During manual analysis, both candidates in Figure 2 were classified as evidence for actual protocols. Iterators have *RemovalPermitted* and *RemovalNotPermitted* abstract states, transitioned

² The bulk of the work of manual examination was performed by Duri Kim, a masters student. Nels Beckman, a Ph.D. student, performed spot checks on these results.

to and from by the **next** and **remove** methods. **remove** can only be called on instances in the *RemovalPermitted* state. **AbstractUndoableEdit** defines several abstract states but the **undo** method can only be called if an instance is both *Alive* and *HasBeenDone*. The candidate in Figure 3, on the other hand, was not categorized as protocol evidence. The exception is really thrown in response to the state of the argument, not the receiver. Even if we wanted to abstract the concrete state of the receiver to prevent the exception, the only reasonable abstraction would require an infinite number of states.

For every candidate that is manually classified as evidence of a protocol, certain information is recorded and used in the second phase of the study. For each piece of protocol evidence, we record the method in which it appears and the class in which that method appears. These classes are referred to as *evidence classes*. But we consider a larger set of types to be protocol-defining. The methods in which protocol evidence appears, and every method they override or implement are considered to be *protocol methods*. Additionally, any public method that calls a private protocol method is considered to be a protocol method (*if* we determine the private method to be part of the “state check” pattern, described below). Finally, the types declaring each of the protocol methods are known as *protocol types*. When we say in the introduction that 7.2% of types declare protocols, these are the types that we are referring to. As these terms will be used frequently in the rest of the paper, we summarize:

Protocol Methods. The methods containing protocol evidence, any methods they override and, if a method containing protocol evidence is private, any public method that calls it.

Protocol Types. The classes and interfaces containing protocol methods.

Our inclusion of private methods and overridden methods is worth further discussion. Regarding our inclusion of overridden methods, our logic here is that, because of subsumption, any subtype may be known statically as its supertype. When a subtype method is part of an object protocol, overridden methods are also frequently part of a protocol, or at best clients must be aware that some subtypes have usage protocols. Therefore, we want to consider calls to those overridden methods as potential client-side uses of protocols.

This strategy addresses one limitation of the ProtocolFinder, that it cannot detect Java interfaces that define protocols. If the implementing methods of an interface have behavior that the ProtocolFinder recognizes as a protocol, then the overridden interface methods will be added to our list of protocol methods.

In a few cases, we removed overridden methods that were added to the set of protocol methods by this process because we felt that the methods are widely used and not normally considered to be part of a protocol. For example, in the Java implementation of the Kerberos authentication protocol, the **KerberosTicket** class defines a protocol of which its **toString** method participates; if a Kerberos ticket has been destroyed, calling its **toString** method results in an **IllegalStateException**. However, **Object.toString** should not be considered a protocol method since most implementations do not have such

```

java.lang.Runnable.run()
java.lang.Thread.run()
java.lang.Object.toString()
java.util.List.add(int, Object)
java.util.List.remove(int)
java.util.AbstractList.add(int, Object)
java.util.AbstractList.remove(int)

```

Fig. 4. Superclass and interface methods automatically considered to be protocol methods due to a subclass that we *removed* from our list of protocol methods. This was done because these methods are widely used, but their contracts do not imply a protocol.

behavior, and it is so widely used that considering it to be one would result in vastly distorted results. (In such situations, one may reasonably conclude that behavioral subtyping was broken.) Figure 4 contains the full list of supertype methods that were removed from the list of protocol methods because they do not in general represent protocols. As far as we can tell, no other widely used supertype methods were misclassified in this manner.

We included the public callers of private methods because we noticed a common pattern in many classes we encountered. Private methods cannot be called outside of the class in which they are defined and as a result will never appear as client usage in the second phase of our study. However, many classes have private “state check” methods which verify that the instance is in some particular state. These methods are called by multiple public protocol methods as a way of avoiding code duplication. For example, the `java.util.PrintStream` class defines a simple Open/Closed protocol, and once the stream has been closed, there are essentially no methods that can be called on the stream. In order to implement this without code duplication, the `PrintStream` method defines a private `ensureOpen` method that is called first thing inside every public method of the class. We want to make sure that we consider those public methods to be protocol methods, even though our analysis does not report them, so we add them when our manual analysis confirms this pattern.

During manual analysis of protocol candidates, two final pieces of data are generated. One of the goals of our study is to determine if object protocols share similar characteristics. Anecdotally, most protocols seem to be rather simple, and somewhat similar (e.g., Open/Closed, Initialized/Uninitialized) and we wanted to determine if this was generally true. While manually examining each potential protocol, we did our best to observe similarities and group them into categories based on these similarities, using single coding. Rather than defining the categories a priori, we constructed them as new similarities were observed.

Lastly, we are very interested in whether or not protocols are used in multi-threaded applications. We would like to understand the relevance of protocol checkers that work even in the face of concurrency, such as our own work [4] and that proposed by Joshi and Sen [17]. So, for each piece of protocol evidence, we recorded whether synchronization primitives (e.g., locks, monitors) were used in the surrounding code. Such use indicates that the class has been designed to be used on multithreaded programs.

Phase 2: Finding Protocol Usage. In the second phase of the study, we examined how often the protocols we discovered in the first phase were actually used. The input of this phase is the list of protocol methods and protocol types generated in the preceding phase. After running an automated analysis on a suite of code, we were left with a list of all classes that called protocol methods as well as a list of all classes that have fields whose types are protocol types, and an estimate of the number of those classes that pass their fields' protocols along to their clients. The static analysis itself is rather simple.

ProtocolUsage. ProtocolUsage is a flow-insensitive static analysis. It proceeds by visiting every method call site in a given code-base. At every method call site, regardless of the receiver, the method binding is statically resolved, and the method's fully qualified name is noted. If the method is in the list of protocol methods, a report is issued, *unless* the method call site is inside the same class as the protocol method being called. Such a call would more accurately be described as an internal interaction rather than a client-provider interaction.

Note that if a class calls protocol methods of its super-class this is considered to be an client interaction with a protocol-defining class, even though at run-time there is only one object. A sub-class can validly be considered to be a client of its super-class, in the sense that a programmer extending another class must be aware of and understand the super-class's rules of use.

ProtocolUsage also looks for instance fields whose types are protocol-defining. In this part of the analysis, at every field declaration, the field's type is resolved. If this type is contained in the list of protocol types, a report is issued.

We are interested in fields of protocol type because they may potentially represent an even closer level of interaction with a protocol-defining type. Since objects referenced by fields are in the heap and may be accessed at any time by member methods, it is more difficult for programmers to obey their protocols than objects that are simply passed and returned amongst methods. Additionally, in our experience it is often the case that classes with fields that define protocols expose those protocols to their own clients.

Manual Examination. While we did not have the time to investigate all of the fields of protocol type, we did want to get an estimate of the number of classes acting as protocol wrappers, passing on the protocols of their fields to their clients. To this end we took a random sample of the classes containing fields of protocol types (approximately 7%) and we manually investigated those classes to see whether or not the protocols of the fields were passed on to their classes. We recorded whether or not this was the case, and used the rate of protocol passing-on to get a rough estimate for the entire suite of phase two programs.

This is the end of the second phase of our study.

2.3 Programs Under Analysis

We ran the ProtocolFinder tool on four open-source programs, in order to find out how many protocols they defined. We then ran the ProtocolUsage analysis on those four plus twelve additional programs to determine how often code acts as a client to protocol-defining code.

All of the programs we analyzed in both phases are shown in Table II, along with their sizes descriptions. With the exception of the standard library and our own analysis framework, Crystal, they all come from the Qualitas Corpus [24]. We attempted to select relatively large, popular open-source programs, and to have a mix of library/framework software as well as end-user applications. By choosing a wide variety of programs we reduce the risk that the programs we analyzed contain abnormally many (or few) protocols. The desire to include both libraries/frameworks and end-user applications is based on our own intuition. We hypothesized that libraries and frameworks are more likely to define types with object protocols since they may be wrapping some underlying system resources that is inherently stateful (e.g., sockets and files).

During the course of the study we examined 1.9 million lines of Java, of which 1.2 million was used in the first phase of the study, and of that portion, one million of which is the Java standard library. Examining the Java standard library for object protocols was a high priority. Because of its wide use in most Java programs, knowing which types in the standard library define protocols enables us to analyze client usage of protocols in many more programs. In fact almost all of the client-side protocol usage in our study was usage of standard library types. This makes sense since, for example, Ant is unlikely to use any protocols defined in PMD, Azureus or JDT and we do not know any of the protocols it defines, since it was not part of the first phase of our study.

2.4 Risks

There are a number of potential risks and threats to validity in the study as designed. Here we discuss some of those risks, as well as mitigating factors.

Some of the most interesting risks in our study are due to our use of static analysis. The use of static analysis is motivated by our desire to examine as large a corpus of programs as possible. Unfortunately, this means the study is subject to the false negative and false positive rates of our static analysis, particularly the ProtocolFinder. For the ProtocolFinder, false negatives are instances where the analysis is run on a piece of code that defines an object protocol and yet the analysis does not report a candidate. False positives are the protocol candidates that are not classified as protocol evidence. False positives are mitigated by manual inspection. Every candidate reported by the ProtocolFinder has been manually inspected to determine whether or not it represents protocol evidence.

However, we can imagine several potential sources of false negatives. The first source is that the ProtocolFinder can only investigate code, and that code must be written in Java. This rules out protocols that are defined by Java interfaces, which contain no code, and native methods, which are written in other languages. We mitigate the former case with our inspection process: when a method is determined to be a protocol method, we note the supertype methods it overrides and add them to our list of protocol methods for use in subsequent phases of the study. For native methods, though, there is not much that we are able to do. Still, of the 120,085 methods we analyzed in the first phase of the study, only 739 were native methods, suggesting that we might not be missing much.

Table 1. The programs analyzed as part of this study, along with their sizes and descriptions

Program	L/F or App.	Version	LOC	Classes (Interfaces)	Description
Phase I: Programs analyzed for protocol definition and usage.					
JSL	L/F	jdk1.6.0_14	1,012,860	8,485 (1,761)	Java standard library
PMD	A	3.1.1.0	26,586	396 (27)	Static analysis
Azureus	A	3.3.2	102,119	900 (354)	BitTorrent client
Eclipse (JDT core)	L/F	3.3	99,691	300 (41)	IDE Framework
Phase II: Additional programs analyzed for protocol usage.					
ant	A	1.7.1	91,679	962 (71)	Build tool
antlr	A	2.7.7	41,880	186 (35)	Lexer/parser tool
aoi	A	2.5.1	81,597	438 (26)	3D modeler
columba	A	1.0	68,267	982 (109)	GUI email client
crystal	L/F	3.4.1	17,052	187 (66)	Static analysis framework
drjava	A	20050814	59,114	639 (79)	Educational IDE
freecol	A	0.7.4	62,641	434 (21)	Civilization clone
log4j	L/F	1.2.13	13,784	178 (16)	Logging library
lucene	L/F	1.4.3	25,472	276 (15)	Text search library
poi	L/F	2.5.1	47,804	417 (28)	Microsoft document library
quartz	L/F	1.5.2	22,171	121 (25)	EJB scheduling framework
xalan	L/F	2.7.0	161,008	1,004 (65)	XSLT XML transformation engine
Total	8×A 8×L/F		1,933,725	15,905 (2,739)	

L/F=Library or Framework A=Application

Another source of false negatives comes from code that does not attempt to detect protocol violations, in other words, protocol-defining code that does not fit the pattern that the ProtocolFinder is looking for. The ProtocolFinder requires code to check or use the value of a receiver field inside a conditional expression and then throw an exception in one branch of the conditional. APIs that fail in an undefined manner when their protocols are violated likely would not fit this pattern. As an example, consider a class defining an initialization protocol, which will throw a null pointer exception if its methods are called before initialization due to null fields. Such a protocol would likely not be detected by our analysis. We believe that well-designed code will generally attempt to detect violations of its own protocols. However, this scenario is likely a source of real false negatives.

Similarly, APIs that define protocols due to their delegation to other, protocol-defining APIs may be missed by our ProtocolFinder. For example, an enhanced

stream that wraps another stream, and delegates calls may define a protocol that is very similar to the underlying stream. While we do not have a direct way of finding these protocols, we are attempting to gauge how likely they might be by reporting the number of classes whose fields themselves define protocols. Then, based on a manual examination of a sample of those classes, we estimate the number of unexamined classes that delegate the protocols of their fields.

Lastly, we have the typical threats of any empirical study: that our selection of programs may be biased, not representative, or too small to draw meaningful conclusions. We have done our best to draw a variety of programs from a respected corpus of popular Java programs [24] that was as large as possible given our time constraints.

3 Results

In this section we present the results of our study³, with little additional discussion. Discussion of the results is postponed until Section 4. The results of running the ProtocolFinder analysis are discussed in Section 3.1, categories of protocols we found are discussed in Section 3.2 and the results of running the ProtocolUsage analysis are discussed Section 3.3.

The summary is that a little over 2.2% of all classes on which we ran our ProtocolFinder define protocols. 7.2% of all types are considered to define protocols when we include supertypes, and approximately 13.3% of all the classes on which we ran our ProtocolUsage analysis use object protocols as clients. 98% of the protocols we found fit into one of seven simple categories.

3.1 Protocol Definitions

Table 2 contains the results of running the ProtocolFinder analysis on the four code bases in phase one. The first column contains the number of candidates reported by the ProtocolFinder analysis. These varied from around 2,600 for the Java standard library to 32 for PMD. The next column shows how many candidates were manually classified as protocol evidence. The next column shows the number of classes containing protocol evidence, followed by a column showing the number of types classified as protocol types. (Recall that our list of protocol types includes classes and interfaces containing methods overridden by methods containing evidence of protocols.) Next, “Thread-Safe Evidence Classes” displays how many classes containing protocol evidence use mutual exclusion. Since we are interested overall in how well our analysis is performing, the next column shows the precision of the ProtocolFinder: the ratio of protocol evidence to protocol candidates. The last two columns show the percentage of classes containing protocol evidence relative to the total number of classes and the number of protocol types relative to the total number of types. The last row displays cumulative values for each column, and percentages recalculated from these sums.

³ All the data gathered during this study can be found at the following location: <http://www.nelsbeckman.com/research/esopw/>

Table 2. The results of running the ProtocolFinder on the four phase one code bases

Program	Protocol Candidates	Protocol Evidence	E.C.	P.T.	T.S.E.C.	Precision	%E.C.	%P.T.
JSL	2,690	613	195	842	54	22.8%	2.3%	8.2%
PMD	32	7	3	10	0	21.9%	0.8%	2.4%
Azureus	136	24	19	32	4	17.6%	2.1%	2.6%
JDT	62	4	4	5	0	6.5%	1.3%	1.5%
Total	2,920	648	221	889	58	22.2%	2.2%	7.2%

T.S.E.C.=Thread-Safe Evidence Classes E.C.= Evidence Classes

P.T.= Protocol Types

3.2 Protocol Categories

Of the 613 candidates in JSL that were manually determined to be protocol evidence, we noticed a number of similarities in their structure and intent. In fact, almost all of them could be characterized in one of seven protocol categories, which we will describe in this section. Due to the means by which our analysis produces candidates, the categories we present are largely categories of errors: conditions under which operation of a class will result in an error. Table 3 summarizes the results for each category. More details on each category can be found in Duri Kim’s masters thesis [18].

Initialization (28.1%). Some types must be initialized after construction time but before the object is meant to be used. In the *initialization* category, calls to an instance method m after construction-time will result in an error unless an initializing method i has been called at least once before. Types may (or may not) allow i to be called multiple times, however, it is a feature of this category that objects cannot become uninitialized after they have already been initialized (i.e., initialization is monotonic).

A typical example of this category is the protocol defined by the Java library class `AlgorithmParameters` in the package `java.security`. After an instance of algorithm parameters is constructed, it is not ready for use until one of its three `init` methods is called. Before initialization, calls to the `toString` method will return null, and calls to `getEncoded` and `getParameterSpec` throw an exception.

Deactivation (25.8%). Some types permit deactivation, after which point instances can no longer be used. In the *deactivation* category, calls to an instance method m will fail after some method d is called on the same instance, and it will always fail for the rest of the object’s lifetime. Like *initialization*, types may or may not permit d to be called more than once.

A typical example is the `BufferedInputStream` in the package `java.io`. Once a stream is closed, no further methods can be called on the stream, and it cannot be reopened. A somewhat more interesting example is `FreezableList` from `com.sun.corba.se.impl.iior`. This is a normal mutable list that, at some point during its lifetime, can be made immutable by calling the `makeImmutable` method. After this point mutating methods, like `remove`, can no longer be called.

(This is in direct contrast to other immutable lists, like those created by `Collections.unmodifiableList`, which are immutable for the entire object lifetime.)

Type Qualifier (16.4%). Some types disable certain methods for the lifetime of the object. In the *type qualifier* category, an object instance will enter an abstract state S at construction-time which it will never leave. Calls to an instance method m , if it is disabled in state S will always fail. This category is so-named since it is similar in spirit to flow-insensitive type-qualifiers [13, 9].

Protocols in this category show two distinct behaviors. In some cases, the abstract state that newly constructed instances inhabit can be set by parameters to the constructor. For example, instances of the `ByteBuffer` type in the `java.nio` package may or may not be backed by a byte array. Whether or not they are depends solely on whether or not a backing array was provided at construction-time. If one was not provided, any calls to the `array` method will fail with a run-time exception. In other cases, the instantiating class itself determines the abstract state that all instances will inhabit, relative to the abstract states defined in a super-type. For example, consider the instances returned from calls to `Collections.unmodifiableList` in the Java standard library. All such instances are *unmodifiable* relative to the super-type `List`, which permits both mutable and immutable lists. In both case, clients must be aware of which methods are enabled.

Dynamic Preparation (8.0%). Certain methods cannot be called until a different method has been called to ready the object. In the *dynamic preparation* category, an instance method m will fail unless another instance method p is called before it. If we think of types in this category as having two states, *ready* and *not ready*, this category is distinguished from the initialization category in that an object may dynamically change from ready to not ready at numerous points in its lifetime (i.e., it is not monotonic).

The most familiar example of this category is the `remove` method on the `Iterator` interface. An iterator’s contract states that the `remove` method cannot be called until `next` has been called, and clients must continue to call the `next` method at least once before each time the `remove` method is called.

Boundary (7.9%). Some types force clients to be sure that an instance is still “in bounds.” In the *boundary* category, an instance method m can only be called a dynamically-determined number of times. Calling m more times will result in an error. Typically such types will provide some method c to clients so that they can determine if an subsequent call to m is safe, although clients are not required to call it. We can abstract this into a finite number of states by having *is in bounds* and *isn’t in bounds* abstract states.

A widely known example of this category is the iterator. In an iterator, the `next` method can only be called if the iterator is at a location in the iterated collection where there are subsequent items. Iterators provide the `hasNext` method allowing clients to check dynamically if this is the case.

Redundant Operation (7.3%). In the *redundant operation* category, a method m will fail if it is called more than once on a given instance.

For an example of this category, consider the `AbstractProcessor` class, located in the `javax.annotation.processing` package. If the `init` method is called more than once, the second call will fail. One might wonder, given the name of the method, why this is not considered to be part of the initialization category. The answer has to do with the fact that our categories are oriented towards errors. In the initialization category, methods on an object will fail if the object has not already been initialized. Here, the failure occurs when the `init` method is called a second time.

Domain Mode (4.8%). The *domain mode* category captures protocols for objects that very closely model a domain. In these objects, various “modes,” which are domain-specific, can be enabled and disabled, which in turn cause certain methods related to those modes to be enabled or disabled.

As an example, consider the `ImageWriteParam` class in the `javax.imageio` package. An image may be written with or without compression. `ImageWriteParam` objects control whether and how compression is used for other image objects. The `ImageWriteParam` class defines several compression modes, “no compression,” “explicit,” and “writer-selected.” The parameter’s `setCompressionType` method can only be called when the parameter is in “explicit” compression mode.

Others (1.9%). Finally, there were a smattering of protocols that did not fit any of the previously-mentioned categories, although even these protocols themselves have certain similar characteristics. As examples, we encountered a few instances of types that defined methods that must be called in strict alternation (a single call to method A enables a single call to method B and vice versa). We also found a limited number of protocols that we would describe as *lifecycle* methods, where a type defines more multiple abstract states through which an object transitions monotonically during its lifetime. For example, the `GIFImageWriter` and `JPEGImageWriter` classes in the Java imageio library seem to have this behavior. While we did not encounter many lifecycle protocols, our own experience with Object-Oriented frameworks suggests that they may be more common elsewhere.

3.3 Protocol Usage

Table 4 shows the results of running the ProtocolUsage analysis on the sixteen candidate programs from phase two of the study. The goal here is to see how often classes act as clients of other protocol-defining types. The table contains the following information: The first column after the list of programs is the number of classes in that program that contain calls to protocol methods. The next column shows the percentage of classes in each program that use protocol methods. These numbers range from 4% of all classes using protocols, on the low end, to 28% of all classes on the high end. The next two columns show the number and percentage of classes that have fields whose types are protocol-defining types.

Table 3. Categorization of each of the 648 reports issued by the ProtocolFinder that were evidence for actual protocols

Category	Protocol Evidence	%
Initialization	182	28.1%
Deactivation	167	25.8%
Type Qualifier	106	16.4%
Dynamic Preparation	52	8.0%
Boundary	51	7.9%
Redundant Operation	47	7.3%
Domain Mode	31	4.8%
Others	12	1.9%

Table 4. The results of running the ProtocolUsage analysis on the sixteen candidate code bases

Program	Classes Calling Protocol Methods	% Classes w/ Prot.	% Prot. Fields	% Exposes Protocol Rate	Est. Classes From Total	
JSL	1012	12%	1082	13%	157	
PMD	85	22%	29	7%	0	
Azureus	198	22%	763	8%	31%	234
JDT	13	4%	18	6%	0%	0
ant	269	28%	187	19%	20%	37
antlr	20	11%	16	9%	0%	0
aoi	25	6%	37	8%	0%	0
columba	120	12%	246	25%	8%	18
crystal	9	5%	2	1%	0%	0
drjava	49	8%	107	17%	0%	0
freecol	94	22%	117	27%	0%	0
log4j	39	22%	32	18%	0%	0
lucene	30	11%	27	10%	0%	0
poi	41	10%	13	3%	100%	13
quartz	16	13%	10	8%	0%	0
xalan	91	9%	142	14%	13%	17
Total	2111	13%	2141	13%	17%	356
W/O JSL	1099	15%	1059	14%	18%	196

The column, “Exposes Protocol Rate” shows the percentage of the classes with protocol fields that were found to expose the protocols of those fields to their own clients, of the 7% of classes with protocol fields that we sampled. The column, “Est. Classes From Total” is an estimate of the total number of classes that expose protocols defined by their fields based on this rate. The last two rows show the totals and cumulative percentages for the entire suite, as well as the numbers excluding the Java standard library.

We were also interested in finding out which protocol methods were being called most frequently, and Table 5 summarizes this information. This table contains a list of the fifteen most frequently called protocol methods. During our

examination of the sixteen open-source code bases used in phase two, we found 7,645 calls to protocol methods. We took all the protocol methods that were called, and ordered them by how many times they were called. Table 5 shows the 15 most frequently called protocol methods along with the number of times that method was called in our candidate programs and the percentage of the 7,645 protocol method calls that particular method constitutes. For example, the `next` method of the `Iterator` interface was the most-frequently called protocol method in our study. Of the 7,645 calls to protocols we found, over 2,200 were calls to `Iterator.next`, almost 30% of the calls.

4 Discussion

After running our experiment, we noticed some interesting results. Protocols were defined with small, but significant, frequency and almost all of those protocols fit within a small number of categories. All of the protocols we expected to find we did find, which gives us some confidence in our approach. And a significant number of classes in our study use protocols as clients, even though almost all of the protocols we were looking for were defined in the Java standard library. Interestingly, but not surprisingly, there are a few protocols that are much more widely used than others.

4.1 Sanity Check

As discussed in Section 2.4, we were curious about the ProtocolFinder’s false-negatives: protocols that were defined in the code under analysis but not discovered due to the design of the analysis. One quick sanity check we can do is to make sure that all the protocols we already know about are found by our analysis. This is not perfect, since our ProtocolFinder was designed with these protocols in mind. Still, it is somewhat comforting to see that all of the protocols we have encountered in our own work, and in similar works are found by our analysis.

We expected to see sockets, files, streams and iterators in our results, since those types are widely discussed in related work. And with the exception of the actual `java.io.File` class, which does *not* define a protocol, we were not disappointed. Socket, Readers, Writer, Streams and all their related classes did turn up in our analysis. (Interestingly, `ZipFile` does define an Open/Closed protocol.) We were also previously aware of the `Throwable` and `Timer` protocols.

Additionally, we were happy to see that well-known protocol-defining interfaces, like `Iterator`, were discovered through our process, since, for interfaces, the ProtocolFinder has no code to examine.

4.2 Widely Used Protocols

We were quite interested, although not surprised, by the fifteen most frequently called protocol methods, shown in Table 5. The iterator protocol, examined in

Table 5. The 15 most-frequently called protocol methods, out of a total of 7,645 calls to protocol methods, and percentage occurrence of each method relative to the total

Method	Calls	% Calls
java.util.Iterator.next()	2226	29.11%
java.util.Enumeration.nextElement()	1022	13.37%
java.lang.Throwable.initCause(Throwable)	850	11.12%
org.w3c.dom.Element.setAttribute(String,String)	460	6.02%
java.util.Iterator.remove()	211	2.76%
java.io.Writer.write(int)	182	2.38%
java.io.OutputStream.write(int)	165	2.16%
java.io.InputStream.read()	162	2.12%
sun.reflect.ClassFileAssembler.cpi()	138	1.81%
org.omg.CORBA.portable.ObjectImpl._get_delegate()	90	1.18%
java.io.InputStream.read(byte[],int,int)	89	1.16%
java.util.ListIterator.next()	80	1.05%
java.io.Writer.write(char[],int,int)	77	1.01%
java.io.PrintWriter.flush()	76	0.99%
java.io.OutputStream.flush()	75	0.98%

several recent works [5, 20], appears at the top of the list, and the `next` method of the iterator protocol accounts for nearly a third of all protocol method calls⁴

While this seems rather uninteresting, it does suggest two points. One, that the time spent evaluating protocol checkers against the iterator interface may be well-spent, since a good iterator-checker can check a large portion of the protocols that are used in practice. Second, all of the calls recorded are actual calls to `Iterator.next`, and not instances of Java 5’s enhanced for loop. While at present, these do represent actual protocol uses, where the client needed to understand the `Iterator`’s protocol in order to use it, one suspects that many of these calls could be replaced by the enhanced for loop, which would dramatically reduce the number of protocol clients we observed. (The same cannot be said for calls to `Iterator.remove`.)

The remaining frequently called methods quickly drop off in the frequency of their use. The most-frequently called list leaves something like forty percent of all protocol method calls off. This suggests that most protocols, like most APIs in general, have a small number of clients. Most of the commonly used protocols are quite recognizable: readers, writers, streams and certain collections defining abstract states. Interestingly, when we remove recognizable types (e.g., streams, sockets, files, iterators, throwables and their subclasses) we found that what was left accounted for 21% of all protocol usage. This means there is still a fair amount of use of non-obvious protocols.

⁴ It is worth noting that the `hasNext` method, which we would generally consider to be part of the `Iterator`’s protocol, does not show up at all in our list of protocol methods. This is due to the fact that the implementations of `hasNext` do not normally partake in protocol violation detection by throwing an exception.

4.3 Protocol Categories

We were pleasantly surprised to discover that a small number of categories (seven) could be used to classify almost all of the protocols that we encountered (98%). This is useful because it suggests a new evaluation criteria for developers of typestate checkers. Unless a typestate checker can verify protocols from each of these seven categories, it is unlikely that it will work on most practical examples. Of course, many of the interesting challenges in protocol verification come from the context in which the protocols are used, for example whether or not the relevant objects are aliased [7]. Still, these categories can help to guide analysis evaluation.

It is also interesting that the categories produced during this study have the flavor of “protocol primitives,” and this may have something to do with how the study was carried out. To illustrate, one may have noticed that none of the categories that we found have more than two abstract states. Yet this does not mean that none of the types we investigated had more than two abstract states. Our study proceeded by investigating each location of interest as determined by the ProtocolFinder. We tried to understand only enough of the implementation to determine whether or not we were seeing evidence for a protocol, the state that the class should be in in order not to have that particular exception thrown, and which state the class is in if the exception is thrown. But classes can have different pieces of a protocol that fit into different categories or even multiple protocol pieces that are all in the same category. As an example of the latter case, consider the `Socket` class in `java.net`. A socket instance can be open or closed, its “write-half” can be open or shut down. Both aspects of the protocol are categorized as *deactivation check* protocols, but if one is to consider the class’ protocol in total, it would have at least four abstract states.

All of this is to say that there may be interesting characteristics shared by protocol-defining types that are not captured by our categories. Coming to a better understanding of protocols at a larger level of granularity, while an interesting topic for future work, is out of the scope of this study.

4.4 Other Observations

A number of other points can be made by examining the results of our study. First, object protocols are relatively common. Without context, 7.2% of the types analyzed may not sound like an enormous amount, but consider that, according to a simple analysis, just 2.5% of the 10,246 types in the Java library define Java “Generic” type parameters, a widely heralded new feature of the language.

One point suggested by the data is that protocol use (13% of all classes) is more common than protocol definition (7.2% of all types). This information suggests that client-side protocol checking may be more important than implementation-side checking. Certain protocol-checking approaches have the ability to verify both the correct use of protocols by clients and the correct implementation of protocols by their providers. Such is the case for the approach presented by Bierhoff and Aldrich [5]. While provider-side checking may be important in

some situations, a good client-side protocol checker may give programmers the most bang for the buck.

In Table 4 we showed that 13% of all classes have fields whose types are protocol types. From the 7% of those classes we manually examined in our random sample, 17% of them were found to expose the protocols of their fields to their clients. Extending this rate to the entire set of classes with protocol fields, we estimate that something like 356 of the classes in the phase two programs define object protocols simply because of the ways in which their fields must be used. This represents about 2% of all of the classes we examined in the entire study, and, if accurate, is a significant increase in the percentage of protocol types.

Of all the classes defining protocols, the percentage implemented with synchronization primitives was significant. Out of 221 classes containing protocol evidence, 58 of them, or 26.2% were designed to be accessed by multiple threads concurrently. If protocol checking is considered an area of research interest, this suggests that those checkers should be designed with multi-threading in mind.

We did not observe conclusively that protocols were more likely to be defined by libraries and frameworks than by applications. However, the Java standard library when considered separately, has a much higher percentage of its types classified as protocol-defining (8% vs. approximately 2%). There could be some truth to the idea that code wrapping underlying system resources is more likely to define protocols. However, given our process of gathering protocol types, it might alternatively suggest that the standard library has a deeper type hierarchy.

For protocol usage, there was some difference observed. In programs that we classified as applications, 17.4% of classes acted as clients of protocol-defining methods. For library and framework code, that rate was 11.4%.

We were interested in the variety of types that define protocols. As evidenced by the small number of protocol categories, these protocols were often quite similar, but in fact the contexts in which they were defined vary greatly. This answered one of the questions that helped to motivate this study: Are there *any* protocol types beyond files, sockets and iterators? We can say, confidently, that the answer is yes. The following list shows just a few of the examples we found:

Security `com.sun.org.apache.xml.internal.security.signature.Manifest`,
`java.security.KeyStore`
Graphics `java.awt.Component.FlipBufferStrategy`,
`java.awt.dnd.DropTargetContext`
Networking `javax.sql.rowset.BaseRowSet`,
`javax.management.remote.rmi.RMIConnector`
Configuration `javax.imageio.ImageWriteParam`,
`java.security.AlgorithmParameters`
System `sun.reflect.ClassFileAssembler`, `java.lang.ThreadGroup`
Data Structures `com.sun.corba.se.impl.ior.FreezableList`, `java.util.Vector`
Parsing `net.sourceforge.pmd.ast.JavaParser`,
`org.eclipse.jdt.internal.compiler.parser.Scanner`

4.5 Future Work

Our study suggests a number of potential avenues for future work. For one, the simple static analysis, ProtocolFinder, developed for this study, while useful, is not sound with respect to our own definition of object protocol. Better analyses will likely find even more protocol definitions in the same code base. Alternatively, widening the definition of object protocol to include more object behaviors, will also likely result in finding more object protocols in the same code base, and a wider definition may be of interest to certain researchers.

As discussed in Section 4.3, our current protocol categories are in some sense “micro-categories:” primitive categories from which larger behavioral patterns might emerge. An interesting task for future work is to examine these larger behavioral entities to see if they share common characteristics.

Finally, even if object protocols are common, an interesting question to ask is whether or not they lead to program defects. Studying the correlation between protocol definition and use in a code base and the quality of that code may help to answer this question.

5 Related Work

The problem of finding classes that define protocols is one of protocol inference, and there has been some work in this area. The two most closely-related studies were done by Weimer and Necula [26] and Whaley et al. [27].

Weimer and Necula [26] performed a study on open-source software that in some ways is similar to ours. In their work, they were looking for violations of resource-disposal protocols. For example, a connection to a database that *must* be closed eventually, ideally as soon as it is no longer needed. They examined over four million lines of open-source Java code and found numerous violations of these sorts of protocols. This study, while quite interesting, differs from ours in a number of ways. First off, their focus was on finding violations of protocols rather than characterizing the nature and use of protocols (correct or otherwise) as we have done. While they did look for protocol violations, they made no systematic attempt to discover automatically the types that define such protocols. Rather, they started their experiments with a known list. Additionally, their notion of protocol and our notion of protocol do not quite overlap. They consider protocols to be instances on which some operation must eventually be performed. The protocols we consider, protocols in which calling a method at the wrong time will lead to an error, are not considered in their work.

Both Whaley et al. [27] and Alur et al. [2] have developed effective tools for statically inferring protocol definition. Whaley et al. [27] present a dynamic and a static analysis for inferring object protocols. Their static analysis is inspired by the same reasoning that ours is, and the description contains an in-depth discussion of the practice of “defensive programming,” which is what we have described here as detection of protocol violations. The dynamic analysis they propose can infer more complex protocols than the static analysis. While our experiments cover some of the same ground as theirs (both examine the Java

standard library) our focus is different. Their primary focus is on the analyses themselves, with the frequency and character of the protocols taking a back-seat. Their largest studies were performed using the dynamic analysis, and so in some ways are not comparable since not all of lines of code are executed during dynamic analysis. Our best estimate is that their study covered approximately 550 thousand lines of source, compared with 1.2 million lines of source covered in phase one of our study. Numbers are only reported for the Java standard library experiment. They report that 81 of 914 classes define protocols. Our experiments for version 1.6.0_14 report that 195 of 8,485 classes define protocols, and show how much the Java standard library has grown since version 1.3.1! Still, their work contains some discussion of the relevant methods and interesting features of these object protocols. Our work contains a more systematic description of the protocols encountered, including a classification of those protocols. Lastly their static analysis seems to be more precise. It can detect protocol violations that result in null pointer exceptions, which ours cannot.

Alur et al. [2] propose a related static protocol detector that also seems to be more precise than ours. They also looked at the Java standard library, albeit just a handful of classes. While either of these static analyses might have made a better candidate for our own study, neither are publicly available.

In fact, a large number of other approaches have been proposed for automatically inferring object protocols, both static and dynamic (e.g., [1, 25, 15, 28], Pradel et al. [21] give a good overview). While a more precise static analysis may help the accuracy of our findings it does not affect our overall conclusions. It is our position that dynamic inference is inappropriate for our needs, since using these analyses requires, at a minimum, test cases to exercise parts of code that use protocols. In our attempt to find as many protocols as possible in as much code as possible, finding test cases has proved to be quite difficult.

6 Conclusion

In this paper we presented an empirical study that examined several popular open-source Java programs. The goal was to determine the true nature of object protocols; how often they are defined, how often they are used, and in what way those protocols are similar. In order to examine as much code as possible, which can help us draw broad conclusions, we developed two static analyses, ProtocolFinder and ProtocolUsage, which help us find where protocols may be defined and where they are used. ProtocolFinder in particular may be subject to false negatives, but regardless was able to find many of the most commonly discussed object protocols.

We found that object protocols are occasionally defined (on average, 7.2% of all types were found to define protocols) but more commonly used (on average, 13% of classes acted as clients of protocols). A small number (seven) of rather simple protocol categories were used to classify almost all of the found protocols.

Acknowledgments. Support provided from ARO grant #DAAD19-02-1-0389 to CyLab, and CMU|Portugal Aeminium project #CMU-PT/SE/0038/2008.

References

1. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: ESEC-FSE 2007, pp. 25–34. ACM Press, New York (2007)
2. Alur, R., Černý, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL 2005, pp. 98–109. ACM Press, New York (2005)
3. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
4. Beckman, N.E., Bierhoff, K., Aldrich, J.: Verifying correct usage of atomic blocks and tpestate. In: OOPSLA 2008. ACM Press, New York (2008)
5. Bierhoff, K., Aldrich, J.: Modular tpestate checking of aliased objects. In: OOPSLA 2007, pp. 301–320. ACM Press, New York (2007)
6. Bierhoff, K., Aldrich, J.: Lightweight object specification with tpestates. In: ESEC-FSE 2005, pp. 217–226 (September 2005)
7. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical API protocol checking with access permissions. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 195–219. Springer, Heidelberg (2009)
8. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. SIGPLAN Not. 36(5), 59–69 (2001)
9. Dunfield, J., Pfenning, F.: Tridirectional typechecking. In: POPL 2004, pp. 281–292. ACM Press, New York (2004)
10. Fahndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA 2007, pp. 337–350. ACM Press, New York (2007)
11. Fairbanks, G., Garland, D., Scherlis, W.: Design fragments make using frameworks easier. In: OOPSLA 2006, pp. 75–88. ACM Press, New York (2006)
12. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective tpestate verification in the presence of aliasing. ACM Trans. Softw. Eng. Methodol. 17(2), 1–34 (2008)
13. Foster, J.S., Johnson, R., Kodumal, J., Aiken, A.: Flow-insensitive type qualifiers. ACM Trans. Program. Lang. Syst. 28(6), 1035–1087 (2006)
14. Gopinathan, M., Rajamani, S.K.: Enforcing object protocols by combining static and runtime analysis. In: OOPSLA 2008, pp. 245–260. ACM Press, New York (2008)
15. Heydarnoori, A., Czarnecki, K., Bartolomei, T.T.: Supporting framework use via automatically extracted concept-implementation templates. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 344–368. Springer, Heidelberg (2009)
16. Jaspan, C., Aldrich, J.: Checking framework interactions with relationships. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 27–51. Springer, Heidelberg (2009)
17. Joshi, P., Sen, K.: Predictive tpestate checking of multithreaded Java programs. In: ASE 2008, pp. 288–296 (September 2008)
18. Kim, D.: An empirical study on the frequency and classification of object protocols in Java. Master’s thesis, Korea Advanced Institute of Science and Technology (2010)
19. Lam, P., Kuncak, V., Rinard, M.: Generalized tpestate checking using set interfaces and pluggable analyses. SIGPLAN Not. 39(3), 46–55 (2004)
20. Naem, N.A., Lhotak, O.: Tpestate-like analysis of multiple interacting objects. In: OOPSLA 2008, pp. 347–366. ACM Press, New York (2008)

21. Pradel, M., Bichsel, P., Gross, T.R.: A framework for the evaluation of specification miners based on finite state machines. In: ICSM (2010)
22. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL 2009, pp. 53–65. ACM Press, New York (2009)
23. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* 12(1), 157–171 (1986)
24. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: A curated collection of Java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference (APSEC 2010) (December 2010) Corpus version 20090202r
25. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: ESEC-FSE 2007, pp. 35–44. ACM Press, New York (2007)
26. Weimer, W., Necula, G.C.: Finding and preventing run-time error handling mistakes. *SIGPLAN Not.* 39(10), 419–431 (2004)
27. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: ISSTA 2002, pp. 218–228. ACM Press, New York (2002)
28. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and recommending API usage patterns. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 318–343. Springer, Heidelberg (2009)

The Beauty and the Beast: Separating Design from Algorithm

Dmitrijs Zaparanuks and Matthias Hauswirth

University of Lugano, Lugano, Switzerland
{zaparand,Matthias.Hauswirth}@usi.ch

Abstract. We present an approach that partitions a software system into its algorithmically essential parts and the parts that manifest its design. Our approach is inspired by the notion of an algorithm and its asymptotic complexity. However, we do not propose a metric for measuring asymptotic complexity (efficiency). Instead, we use the one aspect of algorithms that drives up their asymptotic complexity – repetition, in the form of loops and recursions – to determine the algorithmically essential parts of a software system. Those parts of a system that are *not* algorithmically essential represent aspects of the design. A large fraction of inessential parts is indicative of “overdesign”, where a small fraction indicates a lack of modularization. We present a metric, *relative essence*, to quantify the fraction of the program that is algorithmically essential. We evaluate our approach by studying the algorithmic essence of a large corpus of software system, and by comparing the measured essence to an intuitive view of design “overhead”.

1 Introduction

Given today’s large software systems, consisting of tens or hundreds of thousands of classes, wouldn’t it be nice to be able to automatically distinguish between their essential and non-essential parts? More specifically, wouldn’t it be nice to be able to quantify the amount of algorithmically essential code and the amount of code that primarily serves design? And wouldn’t it be nice to have a tool that automatically locates the essential code? In this paper we present an approach towards this goal.

A software system implements a set of interacting algorithms to achieve a function. As software systems have gotten more complex, software developers have tried to find techniques to combine and implement the algorithms in such a way as to not only achieve efficiency of the running code, but also structure in the software code. The structure is generally referred to as design. There has been a lot of effort to develop effective design methods. Many of them are based on Parnas’ fundamental principle of “information hiding” [13]. The expectation is that good design will impose an overhead (often in the form of extra indirection) during execution, but that it will provide benefits in terms of understandability and maintainability of the software.

In his seminal “No Silver Bullet – Essence and Accident in Software Engineering” [3], Brooks provides an *intensional* definition of essential complexity (“complex conceptual structures that compose the abstract software entity”) and accidental complexity (“*representation* of these abstract entities in programming languages”). His definition of essential complexity refers to the algorithms to be implemented in a system. Any implementation of such algorithms in a given programming language will necessarily entail design decisions such as: Do I extract this code into a separate method? Do I implement this traversal iteratively or recursively? Do I use polymorphism or a switch statement? Do I introduce a facade? Do I use delegation or inheritance? Do I use a visitor pattern or do I place the computation in the structure itself? All of these design decisions affect the resulting code. None of them (should) affect the essential algorithms implemented by the system.

Our goal is to identify the essential parts of a software system by analyzing its code. To automate this task, we have to provide an *operational* definition of essence. We do this by focusing on *repetitions*. Repetitions are essential for the algorithmic complexity of a program. If there are no repetitions, the asymptotic computational complexity is constant, or $O(1)$. While *conditionals* affect computational complexity, adding a conditional statement to a structured program can only reduce its computational complexity. Thus, to keep our analysis tractable, we focus on the most important algorithmic aspect in code: repetitions.

In most programming languages, there are two forms of repetition: loops and recursion. *Loops* correspond to cycles in the *control flow graph* of a method, while *recursions* correspond to cycles in the *call graph* of a program. Our analysis thus identifies the loops in all methods and it determines the recursive cycles in the program’s call graph. The duality between loops and recursions allows us to use the same algorithm for both analyses.

More precisely, our analysis creates *forests of nested cycles* for each of these graphs. For control-flow graphs, such a forest is known as a “loop nest tree”. A cycle is a set of nodes so that each node in the set is reachable from each other node in the set. Moreover, each cycle has a set of *header nodes*, through which execution can enter the cycle. In control-flow graphs directly compiled from structured programs, loops contain only a single header node. However, loops in control flow graphs of unstructured programs, and recursive cycles in the call graph, can have multiple headers.

Our approach to determine the algorithmically essential parts of a program identifies all header nodes of iterative and recursive cycles. Nodes in cycles that are not headers are not deemed algorithmically essential. This relatively straightforward approach allows us to identify the (algorithmically inessential) methods introduced by design decisions such as the use of collections, facades, iterators, encapsulation, or visitors.

The result of our analysis is a representation we call the *loop call graph*, in which we highlight essential parts of the program. Moreover, based on that representation, we compute metrics that quantify the algorithmic essence of a system (or subsystem) as well as its design “overhead”.

1.1 Rationale

Our approach is based on the following design rationale:

Localizable. We do not just want to have a global overall metric measuring essence, but we want an approach that explicitly identifies essential and inessential parts in programs.

Intuitive. The program parts identified as essential need to correspond to a programmer’s intuition of what is essential in the underlying algorithm.

Stable. When analyzing two implementations of the same algorithm, the essential parts in both implementations should correspond to each other.

Language-independent. Our approach should not depend on a specific programming language. Two equivalent programs written in different languages should result in equivalent essential parts.

The remainder of this paper is structured as follows. Section 2 presents a motivating example. Section 3 explains our approach and analysis. Section 4 discusses the tool that implements our approach for Java programs. Section 5 uses our approach to characterize a large body of Java programs. Section 6 discusses the relationship between essence and design. Section 7 connects essence to related work. Section 8 discusses usage scenarios for our metric and limitations of our approach, and Section 9 concludes.

2 Motivation

Consider the three implementations¹ of the factorial function in Listings 1, 2, and 3. They vary in their amount of indirection: Listing 1 expresses the computation as two nested for loops, Listing 2 factors out the multiplication into a separate method, and Listing 3 introduces separate methods for even more basic computational steps. However, all three implementations perform the same essential computation.

Many developers probably would consider Listing 2 as the best of the three designs. We believe this is because neither the call graph nor the control flow graph is excessively big. Figure 1 shows those two graphs (left and center column) for each of the three listings (top to bottom). Listing 1 has a relatively large control-flow graph, while Listing 3 has a similarly large call graph. Unfortunately, we cannot use this balance between call graph and control-flow graph size as guidance for a design metric: the size of the call graph is unbounded because it grows with the size of the program.

However, if we identify the repetitions (in these iterative programs, the loops), we can use them to measure the amount of algorithmically essential code versus the amount of design-related code. We do this in our new representation, the loop call graph (right-most column in Figure 1), which adds loop nodes into the call graph. Having a single representation allows us to combine information about recursion with information about loops.

¹ For the sake of a simple example, assume the language does not have a multiplication operator, which requires the developer to implement multiplication explicitly.

Listing 1. Too little indirection

```

private static int fac(int n) {
  int f = 1;
  for (int i=1; i<=n; i++) {
    int p = 0;
    for (int j=1; j<=i; j++) {p=p+f;}
    f = p;
  }
  return f;
}

```

Listing 2. Enough indirection

```

private static int fac(int n) {
  int f = 1;
  for (int i=1; i<=n; i++) {f=mul(f, i);}
  return f;
}
private static int mul(int a, int b) {
  int p = 0;
  for (int j=1; j<=a; j++) {p=p+b;}
  return p;
}

```

Figure 1 shows that each of the three implementations contains two loops. The loop call graph shows that, at runtime, the multiplication loop will be nested inside the factorial loop, no matter which implementation we use. Moreover, it shows that the number of inessential nodes in the loop call graphs differs significantly. The *relative essence*, or the number of essential nodes divided by the number of method nodes, changes from 2/1 to 2/2 to 2/5. As these values show, a high relative essence is not necessarily positive: it may indicate the absence of any modularization. However, an essence close to 0 is an indication of “overdesign”, or too much indirection.

3 Approach

Our approach to identifying essential and accidental parts of computation is to look for repetition in the computation. This repetition manifests itself either as loops (cycles in control-flow graphs) or recursions (cycles in call graphs).

To reason about repetition overall, we have to combine information from both of these graphs. In theory, we could build a whole-program control-flow graph. In this way, call graph cycles would result in cycles in the whole-program control-flow graph. However, given the size of modern software, the size of the resulting whole-program control-flow graph would grow too big for efficient analysis.

Listing 3. Too much indirection

```
private static int fac(int n) {
    int f = 1;
    for (int i=1; lessOrEqual(i, n); i=addOne(i)) {f=mul(f, i);}
    return f;
}
private static int mul(int a,int b) {
    int p = 0;
    for (int j=1; lessOrEqual(j, a); j=addOne(j)) {p=add(p, b);}
    return p;
}
private static int add(int a,int b) {return a+b;}
private static boolean lessOrEqual(int a,int b) {return a<=b;}
private static int addOne(int a) {return add(a, 1);}

```

Instead of combining call graph and control-flow graph, we propose a multi-level analysis. We analyze each control-flow graph individually to find loops, and we analyze the call graph to find recursions. Each loop and each recursion header represents a computationally essential part of the program, while all other call graph nodes represent accidental parts. Then we combine the results to present the essential and accidental parts of the program and to measure its essence.

3.1 Overview

Our overall approach can be summarized as follows:

1. Build control-flow graphs
2. Identify loop forests in control-flow graphs
3. Build call graph
4. Identify recursion forests in call graph
5. Combine loop forests & call graph into loop call graph
6. Compute metrics

Building control-flow graphs. Control-flow graphs of programs written in languages supporting exception handling often contain a substantial number of exception edges. We expect the control-flow graphs to include those edges, so that we are able to analyze the whole program, including all its exception handlers.

Identify loop forests in control flow graphs. We use an approach to loop detection that works on arbitrary (even un-structured) control flow graphs. Section [3.2](#) describes that approach in detail.

Build call graph. We expect the call graph to include all feasible call edges (e.g. including those due to polymorphic invocations and calls through function pointers). Naively, this can be satisfied by adding edges between each pair of methods. However, such an extremely conservative approach would lead to large

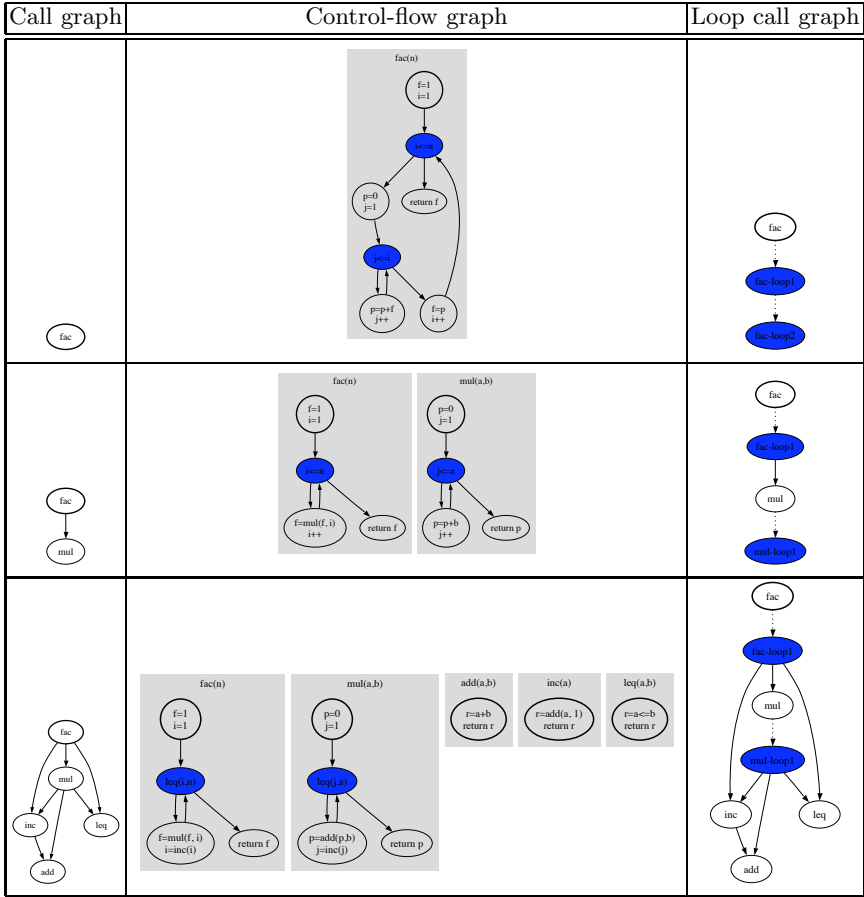


Fig. 1. Call graph, control-flow graph, and loop call graph

numbers of cycles in the call graph and would greatly overstate the amount of recursion in the program.

For strongly-typed languages with polymorphic method calls, efficient analyses such as class-hierarchy or rapid type analysis [11] are able to eliminate most infeasible call edges. For weakly-typed languages, more expensive flow-sensitive analyses or pointer analyses are necessary to get a reasonably precise call graph.

Our recursion detection approach requires one node of the call graph to be denoted as the *entry* node, and it requires all other nodes to be reachable from that node. When analyzing entire programs, this usually corresponds to the main method. When analyzing libraries, we introduce an artificial entry node that points to all public API methods provided by the library.

Sometimes we are interested in analyzing an application but to exclude (some of) the libraries. In that case we do not create any nodes for the excluded library methods. Call sites in the application that point to excluded library methods do

not lead to any call edges. Frameworks, like libraries, are called by the application, but they also call back into the application. When analyzing frameworks in isolation, we create neither of these (incoming or outgoing) call edges, and thus we may miss recursive cycles that cross between application and framework².

Identify recursion forests in call graph. Skiena [16] proposed to identify recursion in call graphs with the purpose of optimizing non-recursive code. He labels all nodes that are involved in a recursion cycle. However, not all methods involved in a recursion cycle are essential for the computational complexity of the program. Analog to loop analysis in control-flow graphs, we want to identify the “headers” of recursion cycles, and we want to properly handle nested recursion cycles. Doing so requires us to determine recursion forests, forests consisting of trees of nested recursion cycles (“loops”). The duality between control-flow graphs and call graphs allows us to use the same approach we use for detecting loop forests in control-flow graphs.

Combine loop forests and call graph into loop call graph. To present the results of our analysis in a single structure, we combine the loops and the call graph into a *loop call graph*. A loop call graph contains two kinds of nodes: method nodes represent methods and loop nodes represent loops. The graph contains two kinds of edges: call edges represent method calls and loop entry edges represent loop entries. Call edges point from a method or a loop to a method (the callee). Loop entry edges point from a method or a loop to a loop. Loop entry edges originating from a loop represent loop nesting (entry of an inner loop from an outer loop). Call edges only originate in a method if the call site is not located in a loop (otherwise they originate in the innermost loop containing the call site).

Figure 2 shows the loop call graphs for four variations of the “best” factorial example in Listing 2. The left-most graph corresponds to the original example, the other graphs replace one or both methods (`fac` or `mul`) with recursive implementations. Loop entry edges are rendered with dotted lines, while call edges are solid. Loop nodes are filled in blue. Method nodes that are recursion headers are filled in red. The method node that corresponds to the graph entry (`fac`) has a bold outline.

Compute metrics. Given the loop call graph, we compute the metrics that quantify the algorithmic essence of the program. First, we define three direct absolute-scale metrics to count the different kinds of nodes in the loop call graph:

$$N_N = |\text{non-recursive method nodes}|$$

$$N_R = |\text{recursive method nodes}|$$

$$N_L = |\text{loop nodes}|$$

Second, we compute the *absolute essence* E , a simple indirect metric that counts the essential nodes in the loop call graph:

$$E = N_L + N_R$$

² We have observed such cycles in frameworks, such as GUI toolkits, that contain recursively invocable event dispatch methods.

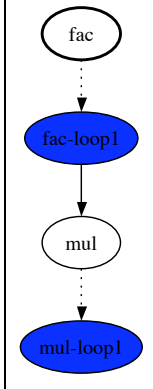
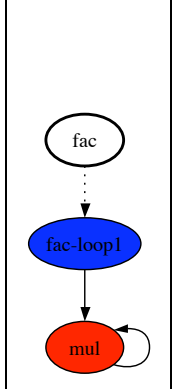
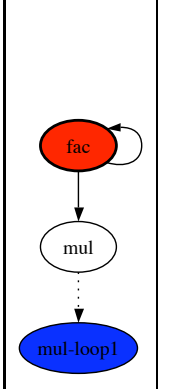
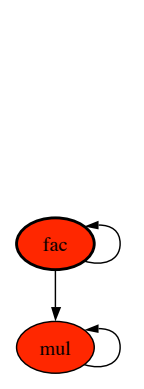
Iterative fac		Recursive fac	
Iterative mul	Recursive mul	Iterative mul	Recursive mul
			
$N_N = 2$ $N_R = 0$ $N_L = 2$ $E = 2$ $e = \frac{2}{2+0} = 1$	$N_N = 1$ $N_R = 1$ $N_L = 1$ $E = 2$ $e = \frac{2}{1+1} = 1$	$N_N = 1$ $N_R = 1$ $N_L = 1$ $E = 2$ $e = \frac{2}{1+1} = 1$	$N_N = 0$ $N_R = 2$ $N_L = 0$ $E = 2$ $e = \frac{2}{0+2} = 1$

Fig. 2. Factorial Loop Call Graphs and Metrics

Finally, we compute indirect ratio-scale metrics to be able to reason about essence independently of program size, culminating in our measure of *relative essence* e :

$$recursiveness = \frac{N_R}{N_N + N_R} \quad loopyness = \frac{N_L}{N_N + N_R} \quad e = \frac{E}{N_N + N_R}$$

Figure 2 shows that E is the same (2) for all four implementations of the program. This means that E is agnostic to the choice between recursive or iterative implementations. Moreover, Figure 1 shows that E is not affected by the design of the program: no matter what degree of indirection is added, E stays the same (2). N_N , however, significantly changes with the design of the program: for Figure 1 it varies between 1 and 5, and it clearly correlates with the amount of indirection introduced by the programmer. Our main metric, e , also stays the same (1) for all four implementations in Figure 2. It does clearly show, though, the differences between the three implementations in Figure 1 (2, 1, and 0.4; for the “underdesigned”, nice, and ”overdesigned” implementations).

3.2 Forest Construction

Two steps in our approach require the construction of forests representing the cycle nesting structure in a graph. We need to find *loop forests* in control-flow graphs and *recursion forests* in loop call graphs.

Reducibility. Loop identification and loop forest construction are standard analyses in optimizing compilers. The control-flow graphs compilers operate on are usually derived from structured source programs, and thus they are usually

*reducible*³ (and even if they were not, optimizers could just decide to skip loop optimizations on irreducible control flow graphs). For this reason, the classic loop identification algorithms used in compilers are unable to properly identify loops in irreducible graphs. Unfortunately, the graphs we encounter in our static analysis, in particular the call graphs (for example those in Figures 5 and 6), are not necessarily reducible. Moreover, unlike a compiler optimization, we cannot just bail out if we encounter an irreducible graph (especially because many realistic call graphs are irreducible, and analyzing the call graph is essential). Thus, we cannot use the classical natural-loop identification algorithms for constructing our recursion forests.

Forests in irreducible graphs. We base our approach on prior work on loop identification in irreducible flow graphs. Ramalingam [14] provides an axiomatic framework for this problem and generalizes prior approaches by Sreedhar et al. [18], Havlak [7], and Steensgaard [19]. Algorithm 1 represents the core of loop forest construction. It iteratively removes specific edges from the graph until the graph contains no more cycles. `FINDSTRONGLYCONNECTEDCOMPONENTS` corresponds to Tarjan’s algorithm, and returns the set of strongly-connected components (the set of loop bodies). In the inner loop, the algorithm iterates over each loop body to determine its header nodes. Unlike in reducible graphs, in irreducible graphs loops can have multiple headers. `IDENTIFYHEADERSSTEENSGAARD` thus returns a set of header nodes for each loop. The algorithm adds each newly identified loop L into the loop forest F . `REMOVELOOPBACKEDGES` in Algorithm 3 then removes all *loopback* edges (edges that point to a header from within the body) from the graph. Finally, the algorithm repeats by again finding strongly-connected components on the now smaller graph.

Header identification. We use Steensgaard’s variant of Ramalingam’s approach, because it produces loops with headers that most closely correspond to the intuitive understanding of the recursion structure in programs. With Steensgaard’s variant, the headers of a loop correspond to all entry nodes (all nodes pointed to from outside the loop). This is more intuitive than Havlak’s or Sreedhar’s variants, which consider subsets of the entry nodes to be headers.

4 Implementation

We implemented a static program analysis tool to measure the essence of Java programs. Our tool uses ASM [12] to statically analyze Java class files. We build control-flow graphs that include all exception edges and use class hierarchy analysis [1] to statically resolve polymorphic call targets. We implemented the forest construction algorithm described in Section 3.2. Our tool determines the number of loops, recursion headers, and total call graph nodes, and computes our metric of essence. It also produces a visualization of the loop call graph with nodes annotated accordingly.

³ In a reducible flow graph [8], each strongly-connected component can only have one entry edge.

Algorithm 1. CONSTRUCTLOOPFOREST

Require: graph $G = \langle N, E \rangle$
Ensure: loop forest $F = \langle \mathbb{L}, \mathbb{C} \rangle$
 {S}CC set S , loop L , loop body B , loop headers H
 {n}ode to loop map $M = \langle N, \mathbb{L} \rangle$
 $S \leftarrow \text{FINDSTRONGLYCONNECTEDCOMPONENTS}(G)$
while $S \neq \emptyset$ **do**
 for all $B \in S$ **do**
 $H \leftarrow \text{IDENTIFYHEADERSSTEENSGAARD}(G, B)$
 $L \leftarrow (H, B)$
 $L_p \leftarrow \text{FINDPARENTLOOP}(F, L, M)$
 $C \leftarrow (L_p, L)$
 $F \leftarrow F \cup \{(L, C)\}$
 $\text{REMOVELOOPBACKEDGES}(G, L)$
 end for
 $S \leftarrow \text{FINDSTRONGLYCONNECTEDCOMPONENTS}(G)$
end while
return F

5 Characterization

To evaluate our new metric, we studied the essence of a large number of Java programs. Our programs consist of the 100 applications of the Qualitas Corpus [20] release 20100719r and the two dominant Java benchmark suites: SPEC JVM [17] release 2008, and Dacapo [2] release 9.12-bach. We included all systems from all three suites. For Qualitas, we used the meta data to indicate which classes to analyze. Instead of including one JRE in the Qualitas corpus, we analyzed JRE 1.6.0 for three different platforms: Linux, Mac OS X, and Windows, because the Java runtime libraries can differ significantly by platform. For SPEC JVM, we analyzed the classes indicated in other characterization studies [21]. For Dacapo, the set of analyzed packages corresponds to the set of packages used when the

Algorithm 2. FINDPARENTLOOP

Require: loop forest $F = \langle \mathbb{L}, \mathbb{C} \rangle$, new loop $L = \langle H, B \rangle$,
 node to loop map $M = \langle N, \mathbb{L} \rangle$
Ensure: return loop L 's parent L_p , or \emptyset if L is a root
 $L_p \leftarrow \emptyset$
for all $n \in B$ **do**
 if $\exists L_m \in \mathbb{L}$ such that $(n, L_m) \in M$ **then**
 $L_p \leftarrow L_m$
 end if
end for
for all $n \in B$ **do**
 $M \leftarrow M \setminus \{(n, L_p)\} \cup \{(n, L)\}$
end for
return L_p

Algorithm 3. REMOVELOOPBACKEDGES

Require: graph $G = \langle N, E \rangle$, loop $L(H, B)$, $H \subseteq B \subseteq N$ **Ensure:** $\forall h \in H \forall b \in B (b, h) \notin E$

```

for all  $h \in H$  do
  for all  $b \in B$  do
     $E \leftarrow E \setminus \{(b, h)\}$ 
  end for
end for

```

Algorithm 4. IDENTIFYHEADERSSTEENSGAARD

Require: graph $G = \langle N, E \rangle$, loop body $B \subseteq N$ **Ensure:** loop headers $H \subseteq B$

```

 $H \leftarrow \emptyset$ 
for all  $n \in B$  do
  if  $\exists n_1 \in N$  such that  $n_1 \notin B \wedge (n_1, n) \in E$  then
     $H \leftarrow H \cup \{n\}$ 
  end if
end for
return  $H$ 

```

benchmarks are run. We label each system with a suffix, according to the corpus it comes from (Q for Qualitas, S for SPEC JVM, D for Dacapo, and we use J for the JREs).

5.1 Size

Our corpus consists of 133 systems, with a total of just over 2 million Java methods and 229536 loops. The systems range in size between 16 and 257562 methods (median: 5132). 159052 of the methods are recursion headers, and there are a total of 38845 recursive cycles. While 31213 (80%) of those recursions have a single header method, 7632 contain multiple headers (and thus lead to irreducible call graphs).

5.2 Essence

The relative essence e of the systems ranges between 0.037 and 0.66 (median: 0.206). Figure 3 plots relative essence versus program size. The logarithmic x-axis shows program size in terms of the number of loop call graph nodes. The highlighted band shows the first, second, and third quartiles of the essence distribution. Half of the systems lie within that band. Systems outside that band are somewhat unconventional.

All the systems with fewer than 200 nodes come from the SPEC JVM suite. Many of them correspond to small, loop-driven algorithm implementations. The fact that our metric separates these systems from “normal” systems in the Qualitas corpus and the Dacapo suite corresponds to the findings of the authors of the Dacapo suite [2].

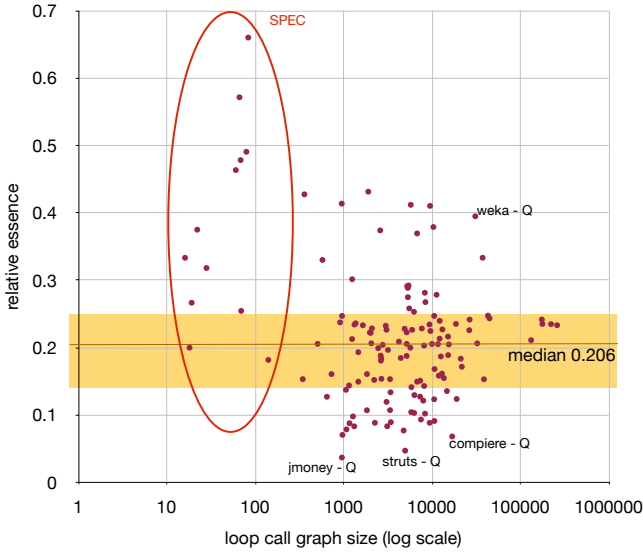


Fig. 3. Essence by program size

The relative essence of *larger* systems usually is closer to the median of the corpus. Large systems, such as the Mac OS X JRE with 219893 methods, are rarely written by a single developer or team, and thus they constitute a mix of code contributions, each contribution with a somewhat different design style. When measuring the essence of the entire system, the different design styles are mixed together and the essence gets close to the median essence of the corpus. For this reason, large systems with a *particularly high or low* relative essence are interesting. Weka is such a large system, with 30589 nodes and a high e of 0.395. Weka is a large collection of machine learning algorithms, and thus its high algorithmic essence is not surprising. JMoney is at the opposite end of the essence spectrum. It is a money management application based on the Eclipse platform. It delegates most repetitions either to existing GUI controls (such as table grids) or collection classes (such as Java’s Arrays.sort).

5.3 Essence by System

Figure 4 presents details about the essence of all the systems in our corpus. We ordered the systems on the x-axis by relative essence. The top chart shows the relative essence, consisting of its two components (loops and recursions). The bottom chart shows the size of the systems in terms of their number of loop call graph nodes. Bars are stacked according to the different node types: non-recursive methods (“normal”), recursive methods, and loops. The y-axis is cropped at 50000, however six systems had well over 50000 nodes (the three JREs, Dacapo’s trade benchmarks, and the Qualitas version of Eclipse).

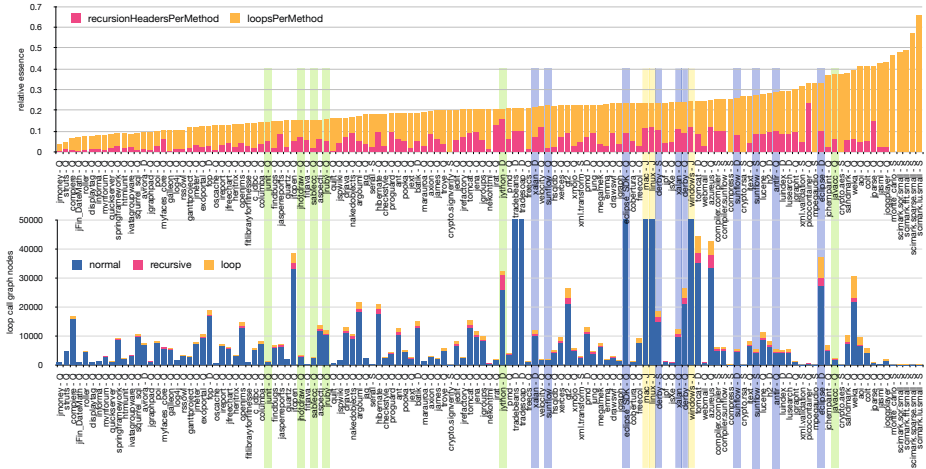


Fig. 4. Essence, recursive methods, and loops in Qualitas, SPEC, and Dacapo systems

Two of the systems in our corpus are traditionally used as good examples of design. JUnit is known for its high design pattern density, and JHotDraw is partially written by the same authors. Those two systems implement dramatically different functionality, however, their relative essence is relatively close (0.144 and 0.153, respectively). Their essence is significantly below the median of 0.206. One could use systems like these, with a design of an exemplary quality, as a reference for the amount of essence a well-designed system should have.

Our corpus contains several groups of related systems. The relative essence of each of the three JREs is above the median. The Windows and Linux versions, which are almost identical, have the same essence of 0.235. The Mac OS X version has a slightly higher essence of 0.242. That version includes additional libraries, such as the QuickTime multimedia framework, a possible source for more algorithm-dominated code. The corpus contains two versions of Derby, one in Qualitas and the other in SPEC. Given the different definitions of what constitutes part of a system, the set of analyzed classes in such cases can significantly differ. For Derby, even though the set of analyzed methods differs, the essence is nearly the same (0.235 vs. 0.242). For Xalan, the difference is relatively similar (0.213 vs. 0.240). For Sunflow, which is part of all three suites, the relative essence differs significantly (0.222, 0.258, 0.275). However, the three suites include significantly different subsets of the Sunflow classes. The essence differs for Eclipse in Dacapo (0.333) and Qualitas (0.233), but Qualitas includes a more recent version of Eclipse and a much larger set of plugins. JRuby (0.158) and Jython (0.206) are two runtime environments for dynamic languages. Jython makes much heavier use of recursion. Finally, SableCC (0.154) and Javacc (0.374) are located almost at the opposite ends of the spectrum. They are both parser generators. The SableCC documentation points out the focus on generating easy-to-maintain, object-oriented, design-pattern-based parsers. This indicates that the

developers of SableCC followed the same design approach when writing SableCC itself. It is interesting to note that the relative essence of SableCC and the well-designed JHotDraw is the same.

6 Essence and Design

In this section we show that essence is directly related to object oriented design concepts, such as code smells, refactorings, and design patterns.

6.1 Essence and Code Smells

Code smells help to identify specific design problems. To understand how code smells relate to essence, we performed two tests.

First, we manually analyzed the commonly known code smells [5], to understand how they correlate with relative essence [4]. Of the 31 smells, 3 strongly indicate low relative essence, 2 weakly indicate low relative essence, 2 weakly indicate high relative essence, and 14 strongly indicate high relative essence. This means that most of the commonly known smells represent issues where the relative essence is too big. This may be an indication that in practice, most problems with software are related to the lack of modularization. Nevertheless, there are some smells (especially “Lazy Class/Freeloader”, “Middle Man”, and “Shotgun Surgery”) that indicate that the relative essence is too low and the amount of indirection too high.

Second, in our corpus of 211507 real-world classes, we manually analyzed those classes that exhibited the highest and the lowest relative essence, to understand their design and to find potential code smells [5]. The 14 classes with the highest relative essence produce mostly “Long Method” smells, often accompanied by “Comments” to break the long method into understandable blocks. In our corpus, 125640 classes (59%) have zero relative essence. Out of that large pool, we picked two small samples. The first zero-essence sample represents the classes with the largest number of methods (and thus method nodes in the loop call graph). This sample, in which each class contains at least 90 methods, represents many “Data Classes” and “Middle Men”. However, it also contains automatically generated classes and adapters with almost empty default method implementations. To avoid the bias possibly introduced by exclusively focusing on classes with many methods, we also investigated a second, random, sample of zero-essence classes. That sample also contains some bad smells like “Data Class”, however many classes in that sample are participants in design patterns, and thus make positive use of indirection.

6.2 Essence and Refactorings

Refactoring is one way to fix the cause of a code smell. In Fowler’s “Refactoring” book [5], Beck differentiates between two kinds of refactorings: those that *add*

⁴ Complete results: <http://sape.inf.usi.ch/essence/algorithmic/smells>

⁵ Complete results: <http://sape.inf.usi.ch/essence/algorithmic/outliers>

indirection where programs are “missing one or more benefits of indirection” (such as for enabling of sharing of logic, for explaining the intent separately from implementation, for isolating change, and for encoding conditional logic through polymorphism), and those that *remove* “parasitic” indirection which isn’t paying for itself (such as left-over intermediate methods, or unused polymorphism). According to Beck, refactoring essentially maximizes the amount of design qualities while minimizing (or at least not unnecessarily increasing) the amount of indirection.

We analyzed how well-known refactorings relate to relative essence, and thus to the amount of indirection⁶. Of the 85 refactorings we studied, only 18 may increase relative essence (14 strongly, 4 of those to a lower degree). A much larger fraction, 47 refactorings, may decrease relative essence (42 strongly, 5 to a lower degree). Three pairs of refactorings most directly relate to relative essence: “Inline Method” and “Extract Method”, “Remove Middle Man” and “Hide Delegate”, and “Replace Delegation with Inheritance” and “Replace Inheritance with Delegation”. For each of these pairs, the first refactoring increases relative essence and the second refactoring (representing the inverse transformation) decreases relative essence. The first two pairs directly introduce or remove delegation. The last pair is a bit more subtle. It transforms between delegation and inheritance. Inheritance could be seen as an implicit form of delegation (dynamic method lookup instead of explicit indirection to the delegate in the application code).

6.3 Essence and Design Patterns

The presence of design patterns in a software system often is considered an indicator of good quality. It is easy to see how design patterns like a “Facade”, an “Adapter”, a “Mediator”, or a “Bridge” introduce extra indirections and decrease the relative essence of software. The effect of some other design patterns on essence are a bit less straightforward. We thus manually analyzed this relationship⁷ for the well-known “Gang of Four”⁶ design patterns. None of the 23 patterns directly implies a higher relative essence. Only 4 patterns are mostly uncorrelated with relative essence. The vast majority (19) represent a design with a low relative essence. Given that design patterns generally reduce relative essence, relative essence could be seen as a measure of design pattern density.

In the remainder of this section we describe the effect of a more complicated pattern, “Visitor”, which is often used in a recursive context. As our running example, assume we want to implement a program that can evaluate abstract syntax trees that represent arithmetic expressions. Algorithm⁵ shows the essence of the solution in the form of pseudo-code. Its relative essence would be $e = N_R/N_R = 1$.

Listing 4⁴ provides a possible implementation of this algorithm in Java, without using the visitor pattern. The tree is represented using a class hierarchy, rooted

⁶ Complete results: <http://sape.inf.usi.ch/essence/algorithmic/refactorings>

⁷ Complete results: <http://sape.inf.usi.ch/essence/algorithmic/patterns>

Algorithm 5. EVAL

Require: n is a tree node, $n.type$ is its type, $n.value$ is its value, $n.left$ and $n.right$ are its children

Ensure: return value = result of evaluating the subtree rooted in n

```

if  $n.type = LITERAL$  then
  return  $n.value$ 
else if  $n.type = ADD$  then
  return  $EVAL(n.left) + EVAL(n.right)$ 
else if  $n.type = SUBTRACT$  then
  return  $EVAL(n.left) - EVAL(n.right)$ 
else if  $n.type = MULTIPLY$  then
  return  $EVAL(n.left) \cdot EVAL(n.right)$ 
end if

```

in an abstract superclass `Node`. The only method of that class, `eval()`, evaluates a node. `Node` has two subclasses, `Literal` which represents a constant value, and `BinOp`, which represents a binary operation. `BinOp` keeps references to its two operand nodes. `Add` is one of the three `BinOp` subclasses, representing an addition. `Subtract` and `Multiply` are analog to `Add`, and are omitted for brevity. In our example, the program entry point is `Main.calc()`, which receives a tree via a reference to its root node and contains a polymorphic call to `Node.eval()`.

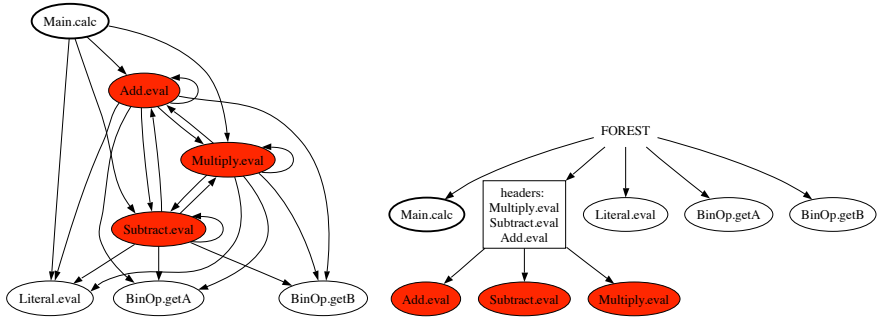


Fig. 5. OO call graph and recursion forest

The top of Figure 5 shows the call graph of this program. The three essential nodes corresponding to the three `eval()` methods represent the headers of one recursive cycle. We call these kinds of recursive cycles “polymorphic recursions”, because their header nodes are implementations of a method declared in a super type, and because they are invoked polymorphically in a recursive way. The three nodes are all header nodes of this single cycle (according to Steensgaard’s approach to header node identification), because there is a node outside the cycle, `Main.calc()`, which points to all of them. If we turned the call in `Main.calc()` into a monomorphic call (e.g. by changing the type of `Main.calc()`’s parameter from `Node` to `Add`), this would change: We would end up with one header

Listing 4. Polymorphic implementation in Java

```

public abstract class Node { public abstract int eval(); }
public class Literal extends Node {
    private int value;
    public Literal(int value) {this.value = value;}
    public int eval() {return value;}
}
public abstract class BinOp extends Node {
    private Node a;
    private Node b;
    public BinOp(Node a, Node b) {this.a = a; this.b = b;}
    public Node getA() {return a;}
    public Node getB() {return b;}
}
public class Add extends BinOp {
    public Add(Node a, Node b) {super(a, b);}
    public int eval() {return getA().eval() + getB().eval();}
}
public class Main {
    public int calc(Node n) {return n.eval();}
}

```

node (`Add.eval`). However, the loop forest algorithm would detect a nested recursive cycle (after eliminating the loopback edges pointing to `Add.eval`), with `Subtract.eval` and `Multiply.eval` as its headers. This means that the number of recursive cycles would change, which is the reason for why we count header nodes and not recursive cycles when computing essence. With our approach, no matter whether we enter this recursive cycle via a polymorphic or a monomorphic call, we end up with three header nodes and the relative essence stays the same⁸: $e = (N_L + N_R)/(N_N + N_R) = (0 + 3)/(3 + 4) = 0.43$. The bottom of Figure 5 shows the recursion cycle forest of this program, which includes a single recursive cycle (rectangle) containing just the three header nodes.

For space reasons, we omit the visitor-based implementation of the traversal. We only show the resulting loop call graph at the top of Figure 6. As we can see, we still have three essential methods (the three concrete elements' `accept` methods). They form the headers of a larger recursive cycle, which also includes the visitor's `visit` methods. Our approach correctly identifies that the visitor pattern introduces an extra level indirection for each method. The number of essential nodes stays the same ($E = 3$), however, the relative essence decreases due to the extra level of indirection: $e = E/(N_N + N_R) = 3/(3 + 10) = 0.23$. Also note that the recursion forest at the bottom of the figure shows the same structure, which means that the visitor does not introduce any nested recursive

⁸ Note that header node identification approaches other than Steensgaard's can violate this property.

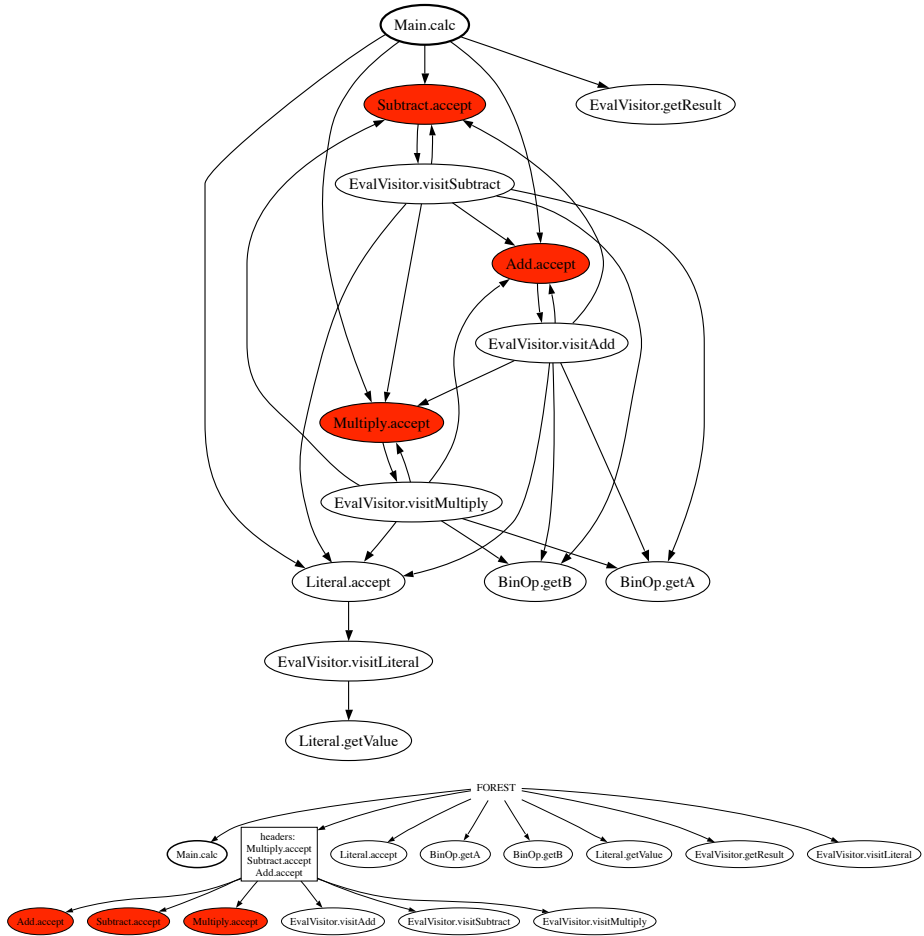


Fig. 6. Visitor call graph and recursion forest

cycles. This corresponds to intuition, because the introduction of the visitor pattern does not change the algorithmic aspect of the program.

7 Related Work

Our approach – and our metric – is related to and inspired by two seminal works in software engineering.

Our operational definition of essence, based on the loop call graph and our two metrics, is similar in spirit to Brook’s intensional definition [3] of essence and accident. We start with the representation of a solution formulated in a concrete programming language, which, according to Brooks, contains essential and inessential aspects. Instead of finding an abstract model representing the

complete essence (and only the essence) of a solution, we aim for the more tractable task of separating the parts of the concrete program that are likely essential from the parts that are likely accidental.

Parnas’ “On the criteria to be used in decomposing systems into modules” [13] describes the concept of information hiding. The tool enabling information hiding is the use of indirection in the form of functions that hide design decisions and internal data structures. Relative essence is our approach towards quantifying the amount of such indirection.

7.1 Conceptual Relationship to Existing Metrics

Essence is a design metric. In this section and the next, we analyze whether essence just represents a different perspective on an existing design metric, or whether essence represents a design property that is not captured by existing metrics. While this section discusses the conceptual relationships between essence and intuitively related metrics, the next section empirically studies the correlation of essence with well-known existing metrics.

Essence is particularly related to four kinds of software metrics: design pattern density, coupling and cohesion, cyclomatic complexity, and bloat.

Design pattern density. Essence is related to Riehle’s design pattern density [15]. Design pattern density determines “which parts of a design are design pattern instances, and which parts are not”. It is defined as “the percentage on an object-oriented framework’s collaborations that are design pattern instances”. Measuring pattern density has many potential benefits, such as estimating the maturity or quantifying the ease of learning a framework. While design pattern density separates the “good” (pattern-based) parts of the design from the “bad” parts of the design, our new metric, essence, separates the overall design from the algorithmic core of a program. A second difference between our essence metric and design pattern density is that our metric is fully automatically computable from code, while design pattern density depends on the prior identification of collaborations and design patterns. While there is much active research in that direction, we are unaware of an approach to automatically and reliably identifying all the necessary collaborations and design patterns in code.

Coupling and cohesion. Essence is also related to coupling and cohesion [4]. While coupling and cohesion tell you what you should *move* around, essence tells you what you might want to *remove*, and it tells you about the absence of indirections you might want to *add*.

Cyclomatic complexity. Essence is related to McCabe’s cyclomatic complexity [10]. Cyclomatic complexity characterizes method bodies consisting of basic blocks, we characterize programs consisting of methods and entire loops. Note that McCabe also defines a measure he calls “essential complexity”, *ev*, which, however, is very different from our notion of essence. His *ev* measures the unstructuredness of a control-flow graph. Structured control-flow graphs, no matter how high their cyclomatic complexity, have *ev* = 1. Thus, essence is closer to McCabe’s cyclomatic complexity, with the difference that we focus on repetitions

(instead of *all* branches) and that we include recursion (which, as our results show, is a major contribution to the complexity of modern software systems).

Bloat. To some degree, essence could be considered a dual to bloat [11]: bloat focuses on unnecessary transformations of *data*, while we identify algorithmically inessential *code*.

7.2 Empirical Correlation with Existing Metrics

The above section discussed the relationship between relative essence and well known metrics on a conceptual level. In this section we empirically determine the correlation of relative essence with design metrics as computed by existing tools.

We believe that relative essence may be a surrogate for *design pattern density*. However, given the absence of an automatic approach to measuring design pattern density, we are not able to validate that hypothesis other than by argument. As we have shown in Section 6.3, the introduction of most design patterns reduces relative essence, and no pattern generally increases relative essence. Thus, at least for the “Gang of Four” patterns, we have reason to believe that pattern density is inversely correlated with relative essence.

Unlike design pattern density, many other design metrics can be computed automatically. However, as Lincke et al. [9] have shown, metric definitions can be ambiguous, and different metrics measurement tools often interpret metrics definitions differently. To make our comparison unambiguous we computed design metrics using four open-source tools: CyVis⁹ for cyclomatic complexity, JDepend¹⁰ for object-oriented design metrics, ckjm¹¹ for the Chidamber-Kemerer [4] metrics, and Dependency Finder¹² for more basic object-oriented metrics. Our study involved 137 metrics. These include metrics computed for each of the 211507 classes, each of the 11018 packages, or each of the 133 applications in our corpus¹³. We found that *none* of the existing metrics are correlated with relative essence. Cyclomatic complexity shows the closest correlation, with a Pearson’s product moment correlation coefficient $r=0.49$. As a rule of thumb, correlation coefficients with an absolute value below 0.6 or 0.7 are considered uncorrelated. Figure 7 shows a scatterplot of cyclomatic complexity versus relative essence. It includes a regression line with a positive slope. Each class corresponds to an individual data point. The figure confirms that the correlation between cyclomatic complexity and relative essence is rather weak.

The second metric, the only other metric with $|r| > 0.4$, is the number of local variables per method. The fact that classes with more local variables per method also show a higher relative essence makes intuitive sense, because methods involving loops or recursion often include local variables.

⁹ <http://cyvis.sourceforge.net/>

¹⁰ <http://www.clarkware.com/software/JDepend.html>

¹¹ <http://www.spinellis.gr/sw/ckjm/>

¹² <http://depfind.sourceforge.net/>

¹³ Complete results: <http://sape.inf.usi.ch/essence/algorithmic/metrics>

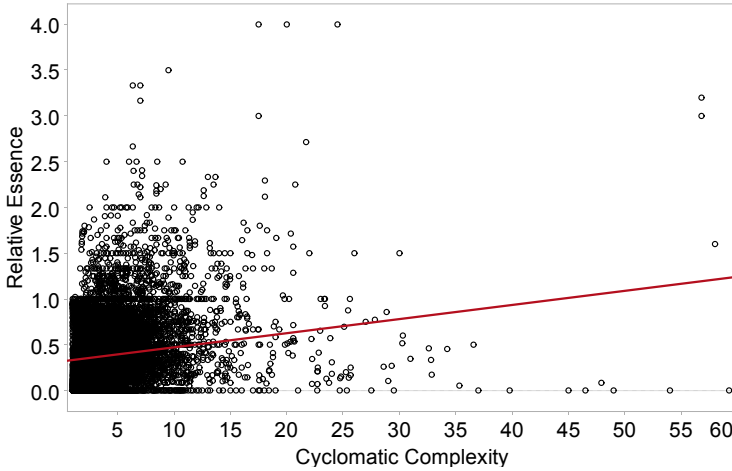


Fig. 7. Relative essence vs. cyclomatic complexity, class granularity: $r = 0.49$

For all other metrics, $|r|$ lies below 0.3. This includes metrics like coupling and cohesion. In particular Chidamber-Kemerer’s LCOM (lack of cohesion of methods) is entirely uncorrelated with relative essence ($r=0.026$).

8 Discussion

In this paper we introduced a novel structure, the loop call graph, and two metrics derived from that structure, absolute and relative essence. We have explained the intuition behind our approach, and we related it to code smells, refactorings, design patterns, and existing design metrics. In this section we discuss potential uses of the loop call graph and essence metrics as well as the limitations of our approach.

8.1 Usage Scenarios

This paper lays the foundation for approaches and tools that analyze and improve systems based on the amount of indirection, modularization, or information hiding in those systems.

Deviation from reference. We believe a good use of our metric is to measure the essence of systems, and to compare it to a reference value of a system of high design quality (such as JUnit and JHotDraw). This is similar to uses of existing design metrics for quality control, where tools produce reports on violations based on a configurable range of admissible metric values. A high essence is indicative of a monolithic design that lacks the structure necessary for good understandability and maintainability. A low essence is indicative of an

“overdesigned” system with excessive indirections, or of a system that consists mostly of glue connecting other systems together.

Problem localization. Besides a mere metric to flag potential problems, our approach also provides a representation (the loop call graph) that helps in locating the sources of the problem. In our own first uses of loop call graphs of student programs, we have found that for small programs (with tens of methods) a node-link diagram visualization, where essential nodes are highlighted, helps to quickly spot clusters of nodes with particularly high or low essence. Given that essence can be computed on any subgraph of the whole-program loop call graph, future approaches could automatically identify connected subgraphs with particularly high or low relative essence, and present these subgraphs to the developer as regions with bad smells. This would avoid the problem of visualizing loop call graphs of realistic programs, with their tens or hundreds of thousands of nodes.

Refactoring recommendation. We can group refactorings into three classes according to their impact on relative essence: refactorings that increase, do not affect, or decrease relative essence. Thus, the relative essence of a given subsystem could guide recommendations on which refactorings to perform. This would help to better modularize those subsystems with abnormally high relative essence, and to eliminate unnecessary indirections in subsystems with abnormally low relative essence.

Quality and process attribute prediction. Is it possible to predict process attributes, such as error rates or times to fix an error, based on our metric? Our hypothesis is that the distance of a system’s essence from the essence of a reference system might be a predictor for various external product quality and process attributes. We would like to study this connection in future work.

8.2 Limitations

We believe that algorithmic essence is a novel concept that promises to lead to several useful approaches and tool. However, the idea of automatically computing essence comes with several important limitations.

Programming language paradigm. While we designed our approach for modern object oriented languages such as Java, it applies to imperative languages in general. However, it is unclear to what degree it applies to functional or logic languages. It would be interesting to study this in future work.

Granularity. We compute relative essence by counting non-recursive methods (inessential nodes) versus loops and recursion headers (essential nodes). If developers aggressively inline methods, they will get a higher relative essence (by eliminating inessential methods while keeping the loops). When thinking on a lower level of abstraction, one could argue that this is not good, because, while the inessential *methods* disappeared, their (inessential) contents still exists (it just got inlined into the body of the top-level method). To circumvent this problem, we could apply our approach at a granularity finer than methods and loops:

Instead of counting methods, we could count statements, or byte-code instructions. We would count loop headers and recursive method call sites as essential nodes, and all other statements (or bytecode instructions) as inessential nodes.

Intent. A main limitation of our approach is the mismatch between Brooks’ *intensional* definition of essence, and our *operational* definition based on loop call graph nodes: (1) In a loop call graph, not all (inessential) non-recursive method nodes represent purely accidental complexity. Clearly, some methods do provide a meaningful algorithmic contribution even if they are not recursion headers and do not contain loops. (2) Not all (essential) recursion headers and loops are necessarily required for solving the problem the program needs to solve. A program may contain unnecessary algorithmic computations, or it may contain overly complex algorithms for a given problem. In general, we believe that *any* operational (and thus automatically measurable) definition of essence will be unable to determine with absolute certainty that a given piece of code is essential according to Brooks’ intensional definition. However, we believe that our approach, while imperfect, is the first to try to quantify this fundamental property, and that it comes close to the intensional definition. By focusing on repetition we focus on the backbone of any algorithm, and we provide (as our results in Section 6 show), a metric that directly relates to design patterns, code smells, and refactorings.

9 Conclusions

While design metrics like coupling and cohesion provide hints for which parts of the system to *move* where, our new metric, essence, provides hints on which parts of the system to *remove*, and where to *add* extra indirections. Essence is a measure of the absence of indirection, layering, encapsulation, or delegation in a system. All of these aspects can positively affect software qualities such as modularity and understandability, however, too much indirection (and thus too little essence) can be an indication of over-engineered systems or even cruft.

Our metric, essence, is an internal product metric. To measure it, we only require the source or binary code of a software system. Essence is simple, precise, and can be measured automatically. It is based on the counts of well known and intuitive concepts: methods, loops, and the headers of recursive cycles in the call graph. Moreover, it can be efficiently computed. It took roughly 3 hours to calculate essence across the entire set of analyzed applications. Essence is a principled metric. It is based on the principle that loops and recursions are the only constructs that can increase the computational complexity of an algorithm. For this reason, any implementation of an algorithm will contain a “backbone” consisting of loops and recursions. By identifying loops and recursions, we can thus identify algorithmically essential parts of a program.

We have studied essence in the largest open corpus of software systems we are aware of, we have found that essence is not correlated to any existing design metrics, and we have found that essence is tightly related to design pattern

density, to code smells, and to refactoring. We hope that essence will prove to be as useful in practice as existing prevalent metrics such as cohesion and coupling.

Acknowledgements. We thank Mehdi Jazayeri, James Noble, Jan Vitek, and the anonymous reviewers for their valuable comments.

References

1. Bacon, D.F., Sweeney, P.F.: Fast static analysis of c++ virtual function calls. In: OOPSLA 1996: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, pp. 324–341. ACM, New York (1996)
2. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, oct 2006, pp. 169–190. ACM Press, New York (2006)
3. Brooks Jr., F.P.: The mythical man-month. (anniversary ed.) Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
4. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20(6), 476–493 (1994)
5. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston (1995)
7. Havlak, P.: Nesting of reducible and irreducible loops. *ACM Trans. Prog. Lang. Syst.* 19(4), 557–567 (1997)
8. Hecht, M.S., Ullman, J.D.: Flow graph reducibility. In: STOC 1972: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, pp. 238–250. ACM, New York (1972)
9. Lincke, R., Lundberg, J., Löwe, W.: Comparing software metrics tools. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA 2008, pp. 131–142. ACM, New York (2008)
10. McCabe, T.: A complexity measure. *IEEE Transactions on Softw. Eng.* SE-2(4), 308–320 (1976)
11. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to java runtime bloat. *IEEE Software* 27(1), 56–63 (2010)
12. ObjectWeb. ASM. Web pages at, <http://asm.objectweb.org/>
13. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15, 1053–1058 (1972)
14. Ramalingam, G.: On loops, dominators, and dominance frontiers. *ACM Trans. Program. Lang. Syst.* 24(5), 455–490 (2002)
15. Riehle, D.: Design pattern density defined. In: OOPSLA 2009: Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp. 469–480. ACM, New York (2009)
16. Skiena, S.S.: Compiler optimization by detecting recursive subprograms. In: ACM 1985: Proceedings of the 1985 ACM Annual Conference on The Range of computing: mid-80’s Perspective, pp. 403–411. ACM, New York (1985)

17. SPEC. SPECjvm2008 (Java Virtual Machine Benchmark), <http://www.spec.org/jvm2008/>
18. Sreedhar, V.C., Gao, G.R., Lee, Y.-F.: Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.* 18(6), 649–658 (1996)
19. Steensgaard, B.: Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research (October 1993)
20. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: Qualitas corpus: A curated collection of java code for empirical studies. In: 2010 Asia Pacific Software Engineering Conference (APSEC 2010) (December 2010)
21. Zaparanuks, D., Hauswirth, M.: Characterizing the design and performance of interactive java applications. In: IEEE International Symposium on Performance Analysis of Systems Software (ISPASS 2010), March 28-30, pp. 23–32 (2010)

The Eval That Men Do

A Large-Scale Study of the Use of Eval in JavaScript Applications

Gregor Richards¹, Christian Hammer¹, Brian Burg², and Jan Vitek¹

¹ Purdue University

² University of Washington

Abstract. Transforming text into executable code with a function such as JavaScript’s `eval` endows programmers with the ability to extend applications, at any time, and in almost any way they choose. But, this expressive power comes at a price: reasoning about the dynamic behavior of programs that use this feature becomes challenging. Any ahead-of-time analysis, to remain sound, is forced to make pessimistic assumptions about the impact of dynamically created code. This pessimism affects the optimizations that can be applied to programs and significantly limits the kinds of errors that can be caught statically and the security guarantees that can be enforced. A better understanding of how `eval` is used could lead to increased performance and security. This paper presents a large-scale study of the use of `eval` in JavaScript-based web applications. We have recorded the behavior of 337 MB of strings given as arguments to 550,358 calls to the `eval` function exercised in over 10,000 web sites. We provide statistics on the nature and content of strings used in `eval` expressions, as well as their provenance and data obtained by observing their dynamic behavior.

*eval is evil. Avoid it.
eval has aliases. Don't use them.*
—Douglas Crockford

1 Introduction

JavaScript, like many dynamic languages before it, makes it strikingly easy to turn text into executable code at runtime. The language provides the `eval` function for this purpose.¹ While `eval` and other dynamic features are a strength of JavaScript, as attested to by their widespread use, their presence is a hindrance to anyone intent on providing static guarantees about the behavior of JavaScript code. It may be argued that correctness and efficiency are not primary concerns of web application developers, but security has proven to be a harder concern to ignore. And, as web applications become central in our daily computing experience, correctness and performance are likely to become more important.

See no Eval, Hear no Eval. The actual use of `eval` is shrouded in myths and confusion. A common Internet meme is that “eval is evil” and thus should be avoided.² This comes with the frequent assertion that `eval` is the most misused feature of the language.³

¹ While JavaScript provides a few other entry points to code injection, such as `setInterval`, `setTimeout` and `Function`, we refer to this class of features as `eval` for much of our discussion.

² <http://javascript.crockford.com/code.html>

³ <http://blogs.msdn.com/b/ericlippert/archive/2003/11/01/53329.aspx>

Although `eval` is a significant feature of JavaScript, it is common for research on JavaScript to simply ignore it [2,11,20,1], claim it is hardly used (only in 6% of 8,000 programs in [8]), assume that usage is limited to a relatively innocuous subset of the language such as JSON deserialization and occasional loading of library code [9], or produce a simple warning while ignoring `eval`'s effects [13]. The security literature views `eval` as a serious threat [19]. Although some systems have unique provisions for `eval` and integrate it into their analysis [7], most either forbid it completely [15], assume that its inputs must be filtered [5] or wrapped [12], or pair a dynamic analysis of `eval` with an otherwise static analysis [4].

True Eval. The goal of this study is to thoroughly characterize the real-world use of `eval` in JavaScript. We wish to quantify the frequency of dynamic and static occurrences of `eval` in web applications. To this end, we have built an infrastructure that automatically loads over 10,000 web pages. For all web page executions, we have obtained behavioral data with the aid of an instrumented JavaScript interpreter. We focus our attention on program source, string inputs to `eval` and other dynamically created scripts, *provenance* information for those strings, and the operations performed by the `eval`'d code (such as the scopes of variable reads and writes). Though simply loading a web page may execute non-trivial amounts of JavaScript, such *non-interactive* executions are not representative of typical user interactions with web pages. In addition to page-load program executions, we use a random testing approach to automatically generate user input events to explore the state space of web applications. Lastly, we have also interacted manually with approximately 100 web sites. Manual interaction is necessary to generate meaningful interactions with the websites.

While we focus on JavaScript, `eval` is hardly unique to JavaScript. Java supports reflection with the `java.lang.Reflect` package, and the class loading infrastructure allows programs to generate and load bytecode at runtime. Dynamic languages such as Lisp, Python, Ruby, Lua, and others invariably have facilities to turn text into executable code at runtime. In all cases, the use of reflective features is a challenge to static analysis. JavaScript may represent the worst case since `eval`'d code can do almost anything.

Our results reveal the current practice and use of reflective features in one of the most widely-used dynamic programming languages. We hope our results will serve as useful feedback for language implementers and designers. The contributions of this paper are:

- We extend the tracing infrastructure of our previous work [18] to record the provenance of string data and monitor the scope of variable accesses.
- We add tools for automatically loading web sites and generating events.
- We report on traces of a corpus of over 10,000 websites.
- We make available a database summarizing behavioral information, including all input arguments to `eval`, and other execution statistics.
- We provide the most thorough study of the usage of `eval` in real-world programs to date.
- We instrumented other means of creating a script at runtime and compare their behavior to `eval`.

Our tools and data are freely available at:

<http://sss.cs.purdue.edu/projects/dynjs>

2 The Nature of Eval

JavaScript, which is a variation of the language standardized as ECMAScript [6], is supported by all major web browsers. It was designed in 1995 by Brendan Eich at Netscape to allow non-programmers to extend web sites with client-side executable code. JavaScript can be best described as an imperative, object-oriented language with Java-like syntax and a prototype-based object system. An object is a set of properties that behave like a mutable map from strings to values. Method calls are simulated by applying arguments to a property that evaluates to a closure; this is bound to the callee. The JavaScript object system is extremely flexible, making it difficult to constrain the behavior of any given object. One of the most dynamic features of JavaScript is the `eval` construct, which parses a string argument as source code and immediately executes it. While there are other means of turning text into code, including the `Function` constructor, `setInterval`, `setTimeout`, and indirect means such as adding `<script>` nodes to the DOM with `document.write`, this paper focuses on `eval` as a representative of this class of techniques for dynamically loading program source at runtime.

The Root of All Evals. `Eval` excels at enabling interactive development, and makes it easy to extend programs at runtime. `Eval` can be traced back to the first days of Lisp [16] where `eval` provided the first implementation of the language that, until then, was translated by hand to machine code. It has since been included in many programming languages, though often under other names or wrapped inside a structured interface.

The Face of Eval. In JavaScript, `eval` is a function defined in the global object. When invoked with a single string argument, it parses and executes the argument. It returns the result of the last evaluated expression, or propagates any thrown exception. `eval` can be invoked in two ways: If it is called directly, the `eval`'d code has access to all variables lexically in scope. When it is called indirectly through an alias, the `eval`'d code executes in the global scope [6, sect. 10.4.2]. All other means to create scripts at runtime, as discussed in Sec. 6, execute in the global scope.

The Power of Eval. JavaScript offers little in the way of encapsulation or access control. Thus, code that is run within an `eval` has the ability to reach widely within the state of the program and make arbitrary changes. An `eval` can install new libraries, add or remove fields and methods from existing objects, change the prototype hierarchy, or even redefine built-in objects such as `Array`. To illustrate the power of `eval`, consider the following example, which implements objects using only functions and local variables.

```
Point = function() { var x=0; var y=0;
  return function(o,f,v){ if (o=="r") return eval(f); else return eval(f+"="+v); }
}
```

Every invocation of the function bound to `Point` returns a new closure which has its own local variables, `x` and `y`, that play the role of fields. Calling the closure with `"r"` causes the `eval` to read the `'field'` name passed as second argument; any other value updates the `'field'`. Calling `eval` exposes the local scope, thus breaking modularity. Exposing the local scope can be avoided by aliasing `eval`, but the global scope is still exposed: any assignment to an undeclared variable, such as `eval("x=4")`, will implicitly declare the variable in the global scope and pollute the global namespace.

Necessary Eval? In modern web applications, the server and client rarely have a persistent connection. Instead, the client makes independent, asynchronous requests every time it needs data. This style of communication is often called Asynchronous JavaScript and XML (AJAX). The data returned is frequently in an application-dependent format, in a portable serialization format such as JSON, or in the form of JavaScript code. If they are in the form of code, then `eval` is the typical means of evaluating this code. Although the canonical means of making such requests is by using an `XMLHttpRequest` (XHR) object, it has the drawback that it is subject to the same origin policy, which prevents requests to a different domain. Many sites divide server functions between different hosts, and as such are forced to use other means which are not restricted by the SOP. Most other means actually evaluate the server response as code regardless.

Until recently, JavaScript did not have its own built-in serialization facility, so `eval` was (and is) often used to deserialize data and code. JSON⁴ is syntax designed to provide a portable way for applications to serialize and deserialize data. JSON is also, by no coincidence, a subset of JavaScript's object, array, string and number literal syntax. An example JSON string is:

```
{ "Image": { "Title": "View from a Room", "IDs": [11,23,33], "Size": { "Height": 125} } }
```

JSON is restrictive; e.g. `{foo:0}` is valid, but `{'foo':0}` or `{foo:0}` are not, though all are semantically equivalent JavaScript expressions. Anecdotal evidence suggests that JSON-like strings that don't adhere precisely to the JSON standard are commonly used by developers. JSON is also commonly `eval'd` along with an assignment to a variable, e.g. `x={foo:0}`. Performing the assignment within the `eval` is unnecessary, as `eval` returns a result. The canonical way to parse JSON with `eval` and assign the result to a variable is `x=eval(y)`.

The use of `eval` is often unnecessary, and is could be replaced by uses of other (less dynamic) features of JavaScript.⁵ Consider the following misuse:

```
eval("Resource.message_" + validate(input))
```

The programmer presumably has some `Resource` object holding a number of messages. To select the right message at runtime, a string such as `"Resource.message_error"` is built out of some user input. To be on the safe side, the input is validated programmatically. Validation is tricky and a large number of code injection attacks come from faulty validators. The above code could be implemented straightforwardly without `eval` as `Resource["message_" + input]`. Rather than invoking the full power of `eval`, the code uses a constructed string to index `Resources`. This achieves the same effect with none of the security risks associated with using `eval`.

The Eval Within. The `eval` function is a performance bottleneck because its mere presence affects how a JavaScript engine can optimize and execute surrounding code. Any optimization performed by the virtual machine must account for the black-box behavior of `eval`. The fact that `eval` can introduce new variables in the local scope means that flexible, deoptimized bytecode must be generated for a function that contains `eval` within its body. This version will always be slower than an equivalent function without `eval`, even if no such variables are actually introduced (see Appendix B).

⁴ <http://www.ietf.org/rfc/rfc4627>

⁵ Recent versions of ECMAScript introduced `JSON.parse` as an alternative to `eval`.

3 Methodology

We now describe the infrastructure and methodology used to collect our data.

3.1 Infrastructure

We quantify usage of `eval` by recording relevant information during JavaScript execution, and subsequently performing offline analyses. The data presented in this paper was recorded using TracingSafari [18], an instrumented version of the open-source WebKit project,⁶ the common web platform underlying Safari, Google Chrome, and other applications. TracingSafari is able to record low-level, compact JavaScript execution traces; we augmented it to also record properties specific to `eval`. In particular, we add provenance tracking for strings, as these might eventually become arguments to `eval`.

TracingSafari records a trace containing most operations performed by the interpreter (reads, writes, deletes, calls, defines, etc.) as well as events for source file loads. Invocations to `eval` save the string argument. Complete traces are compressed and stored to disk. Traces are analyzed offline and the results are stored in a database which is then mined for data. The offline trace analysis component performs relatively simple data aggregation over the event stream. For more complex data, it is able to replay any trace, creating an abstract representation of the heap state of the corresponding JavaScript program. The trace analyzer maintains a rich history of the program’s behavior, such as access history of each object, call sites, allocation sites, and so on.

3.2 Corpus

Gathering a large corpus of programs is difficult in most languages because accessibility to source code and specific runtime configurations is often limited. On the web, this is generally not the case: any interactive web site uses JavaScript, and JavaScript is only transmitted in source form. Furthermore, most websites are designed to function in many browsers.

JavaScript executes in two distinct phases: first, non-trivial amounts of JavaScript are parsed and executed automatically as the result of loading a document in the browser. Further program execution is event-driven: event handlers are triggered by timers and user input events such as mouse movements, clicks, and the like. To capture a wide range of behavior we have compiled a corpus composed of three data sets:

INTERACTIVE	Manual interaction with web sites.
PAGeload	First 30 seconds of execution of a web page.
RANDOM	PAGeload with randomly generated events.

All of our runs were based on the most popular web sites according to the `alexa.com` list as of March 3, 2011. INTERACTIVE was generated by manually interacting with the 100 most popular web sites on the Alexa list. Each session was 1 to 5 minutes long and approximated a “typical” interaction with the web site, including logging into accounts. PAGeload and RANDOM were based on the 10,000 most popular web sites on the Alexa list. PAGeload is intended to record the load-time behavior of pages.

⁶ <http://webkit.org> Rev. 76456

It simply navigates the browser to each page and records execution for a total of 30 seconds without any further interaction. As script execution can recur indefinitely, there is no clear moment when a page has finished loading. In this case, a simple timeout is the most reliable way to include load-time behavior without interaction. RANDOM behaves similarly to PAGeload, but includes a script which will randomly trigger click events on DOM elements with mouse event listeners registered, and click links. One click event is generated per second, for at most 30 events. The final data was recorded between March 3rd and 13th, 2011. All recorded traces are available from our project's site.

These three data sets each cover a useful subset of eval usage in the wild. INTERACTIVE provides the best picture of complete interactions with a web application and is thus the most representative of the usage of eval in JavaScript programs. PAGeload and RANDOM give us breadth of coverage and allow us to study a much larger number of web sites but with a caveat of reduced program behavior coverage. PAGeload will not generate unrealistic behavior, although it may generate atypical behavior. RANDOM can generate unrealistic behavior, but is the best way of obtaining a wide variety of behaviors on a large corpus of sites.

3.3 Threats to validity

Program coverage. As with any tracing-based methodology it is difficult to obtain exhaustive coverage. The problem is compounded by the interactive nature of web applications which are driven by the user interface. Furthermore, as programs are only fed to the browser one page at a time, it is difficult to even assess which fraction of a web site was exercised. Our results may thus fail to uncover some interesting behaviors. This said, we believe that our corpus is representative of typical browsing behavior. Browser versions are fairly easy to ascertain, so it is possible (and common) for JavaScript code to exhibit behavior peculiar to WebKit. Although this does introduce a subtle bias, all other JavaScript implementations introduce comparable bias.

Diversity of programs. Another threat comes from our focus on client-side web applications. It is likely that other categories of JavaScript applications would display different characteristics. For instance, widgets appear to do so [8]. But the importance of web applications and the quantity of JavaScript code on the web mean that this is a class of applications worth studying.

4 Usage Metrics

This section presents a high-level picture of the usage of JavaScript and eval in a broad selection of web pages, as summarized in Table 1. At the time of our study, *all* of the top 100 sites used some JavaScript. For the 10,000 most accessed web sites we found that 89% rely on JavaScript. Similarly, eval was used widely and frequently in our corpus. We have recorded 550,358 calls to eval for a total of 337 MB of string data. Over 82% of the top 100 pages use eval, and 50% of the remaining 10,000 pages do as well. It is noteworthy that the difference in the use of eval between RANDOM and PAGeload is only 2%, which suggests that sites relying on eval do so even without user interaction. On the other hand, the number of *calls* to eval increases significantly in RANDOM.

Table 1. Eval usage statistics

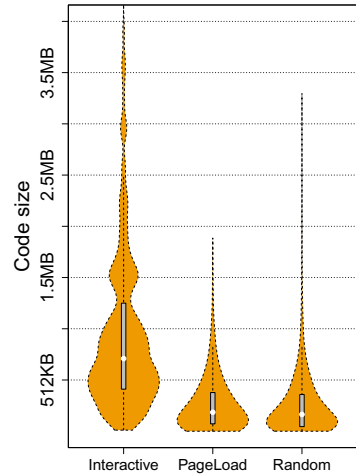
Data Set	JavaScript used	eval use	Avg eval (bytes)	Avg eval calls	total eval calls	total eval size (MB)	total JS size (MB)
INTERACTIVE	100%	82%	1,210	84	7,078	8.2	204
PAGELOAD	89%	50%	655	34	158,994	99.3	1,319
RANDOM	89%	52%	627	61	384,286	229.6	1,823

JavaScript code size. As in our previous study, we found that most web sites have less than 512KB of JavaScript code, with some significant outliers, especially in the most popular sites. Fig. 1 displays the distribution of the total size of the JavaScript code loaded during execution of each website, including source loaded via eval. When the same code is loaded multiple times we only took it into account once. The mean sizes are 973KB for INTERACTIVE, 187KB for PAGELOAD, and 270KB for RANDOM. The largest website was yahoo.com with 5.09MB of JavaScript code. The difference in code size between PAGELOAD and RANDOM is explained by the fact that a mouse click (or any other event) may cause additional code to be loaded.

Number of eval call sites. We observed that the average number of call sites is small, and interactive behavior is correlated with a greater number of call sites. Fig. 2 shows the distribution of the number of direct call sites to the eval function that are reached per session, for sessions where at least one call to eval was made. User interactions frequently uncovered new call sites: while the mean number of call sites is only 1.7 in PAGELOAD, the mean of RANDOM and INTERACTIVE is 4.0 and 13, respectively. The maximum number of call sites in INTERACTIVE was 77, which is lower than both PAGELOAD and RANDOM (127 and 1331 call sites, respectively).

Number of calls to eval. Unsurprisingly, user interaction is correlated with the number of calls to eval, and websites call eval in both phases of script execution. We observed an average of 38 calls to eval in the INTERACTIVE data set, 28 in PAGELOAD, and 85 in RANDOM. Fig. 3 gives the distribution of the number of invocations of eval per website. The largest number of invocations occurs in RANDOM with a whopping 111,535 calls.

Amount of source loaded by eval. The size of source text passed to eval widely varies depending on *what* is being evaluated. Fig. 4 shows the distribution of source text size. Strings range in size from empty strings to large chunks of data or code. While for INTERACTIVE about two thirds of the strings are less than 64 bytes long, the maximum observed size was 225KB. The PAGELOAD and RANDOM data sets tell similar stories, 85% and 80%, respectively, of strings are less than 64 bytes, but they peak at

**Fig. 1. Code size.** The distribution of total size of code loaded during evaluation of each website.

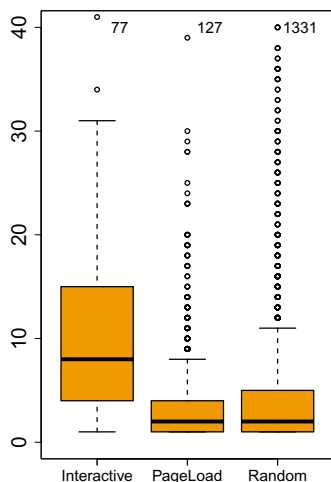


Fig. 2. Eval call sites. The y-axis is the distribution of the number of call sites to the eval function in websites that call the function at least once. (Max value appear on top)

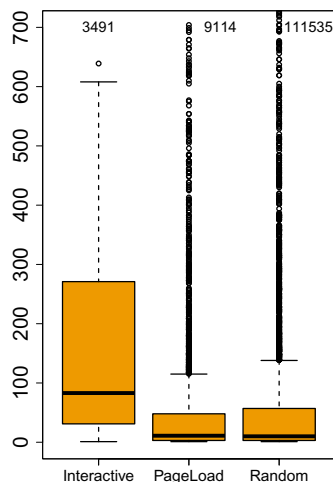


Fig. 3. Eval calls. The y-axis is the distribution of the number of calls to the eval function in websites that call the function at least once.

460KB and 515KB respectively. The average source size is 1,210 bytes for the INTERACTIVE, 655 bytes for the PAGeload, and 627 bytes for the RANDOM runs. JSON in particular carries more data on average than other categories. The average size of JSON strings was 3,091 bytes in INTERACTIVE, 2,494 bytes in PAGeload and 2,291 bytes in RANDOM. However the medians were considerably lower (1,237, 31 and 54 bytes, respectively), which is consistent with the distribution of sizes seen for other categories. The maximum JSON size is 45KB for INTERACTIVE and 459KB for the other data sets.

Amount of computation via eval. With the exception of loading JavaScript libraries via eval, most calls performed relatively few operations. Fig. 5 shows the distribution of eval trace lengths. The trace length is a rough measure of the amount of computational work performed by any given eval. The operations captured in a trace include object access and update, calls as well as allocation. The median number is again low, 4, with the third quartile reaching 10 operations. The spread beyond the third quartile is extreme, with the RANDOM sessions recording traces of up to 1.4 million operations. Given the maximum size of the source strings passed to eval reported in Fig. 4 this size is not too surprising. In contrast, the maximum number for the INTERACTIVE sessions is low compared to its maximum size of source strings.

In all datasets, the largest eval'd strings, both in terms of length and in terms of event count, were those that loaded libraries. In JavaScript, loading a library is rarely as simple as just installing a few functions; tasks such as browser and engine capability checks, detection of other libraries and API's, creation of major library objects and other such initialization behavior constitutes a large amount of computation relative to other eval calls.

Aliasing of eval. We observed that few programmers took advantage of the differing behavior that results from calling an alias of eval. In INTERACTIVE, 10 of the top 100

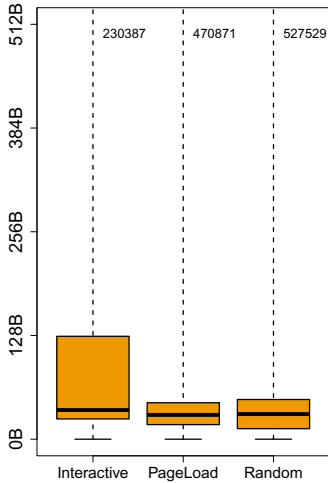


Fig. 4. Eval string sizes. The y-axis is the distribution of the size of `eval` arguments in bytes.

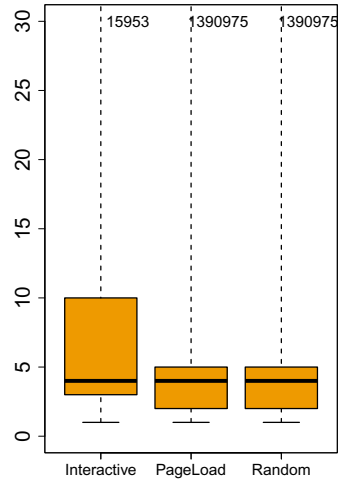


Fig. 5. Events per eval. The y-axis is the distribution of the number of events performed in an `eval`.

sites aliased `eval`, but calls to such aliases accounted for only 0.9% of all `eval` invocations. In `PAGELOAD` and `RANDOM`, only 130 and 157 sites, respectively, used an alias of `eval`, accounting for 1.3% and 0.8% of `eval` strings respectively. Manual inspection revealed use cases where programmers used an alias of `eval` to define a function in the global scope, without realizing that the same effect could be achieved by simply assigning a closure to an undeclared variable. See Appendix C for an illustration.

Presence of JavaScript libraries. In our corpus, JavaScript libraries and frameworks were present on over half of all sites. Table 4 gives the proportion of the sites using common libraries. We found that jQuery, Prototype, and MooTools were used most often. JQuery is by far the most widespread library, appearing in more than half of all websites that use JavaScript. Other common libraries were detected in under 10% of all sites. The Google Closure library used by many Google sites is usually obfuscated, and thus not easily detectable. We do not report on it here. Libraries are sometimes loaded on demand, as shown by the spread between the `PAGELOAD` and `RANDOM` (for instance 53% and 60% for JQuery).

One might wonder if libraries are themselves a major contributing factor to the use of `eval`. Manual code review reveals that `eval` and its equivalents (the Function constructor, etc) are not required for their operation. The only uses of `eval` we have discovered are executing script tags from user-provided HTML strings, and as a fallback for browsers lacking `JSON.parse`. Thus, libraries are not a significant contributor to the behavior or use of `eval`.

Data Set	jQuery	Prototype	MooTools
INTERACTIVE	54%	11%	7%
PAGELOAD	53%	6%	4%
RANDOM	60%	7%	6%

Table 2. Common libraries. Percentage of website loading one of the following libraries: `jquery.com`, `prototypejs.org`, `mootools.net`. We have no data for `code.google.com/closure`.

5 A Taxonomy of Eval

The previous section gave a high-level view of the frequency of `eval`; we now focus on categorizing the behavior of `eval`. We look at five important axes. Firstly, we study the **mix** of operations performed by the code executed from an `eval`. Next, we look at what **scope** is affected by operations inside `eval`'d code. Operations that mutate shared data are more likely to invalidate assumptions or pose security risks than operations that are limited in scope to data created within the `eval`. Thirdly, we try to identify **patterns** of usage. A better classification of the patterns of `eval` usage can help language designers provide limited, purpose-specific alternatives to `eval`, and also provide a better understanding of the range of tasks done within `eval`s. Fourthly, we investigate the **provenance** of the string passed into `eval`. This comes directly from a desire to better understand the problems linked to code injection attacks. Our last axis is **consistence**, or how the arguments to a particular `eval` call site vary from invocation to invocation. We focus on each axis independently, discussing the relationships between them when relevant, then discuss the implications of each on analyses and other systems.

5.1 Operation Mix

The operations recorded in our traces are simplified, high-level versions of the WebKit interpreter's bytecodes. We report on stores (STORE), reads (READ) and deletes (DELETE) of object properties. These include indexed (`x[3]`) and hashmap style (`x["foo"]`) access to object properties. We also report on function definitions (DEFINE), object creations (CREATE), and function calls (CALL). Fig. 6 gives the distribution of operations performed by `eval`'d code for each of our three data sets. The distribution of operation types across the `PAGELOAD` data set is consistent with earlier findings, and suggests that `eval`'d code is not fundamentally different from general JavaScript code. In particular, `eval` is not solely used for JSON object deserialization, as some related work assumes. That said, `INTERACTIVE` sessions do contain a greater proportion of `STORE` and `CREATE` events, which we attribute to JSON-like constructs. We will consider the proportion of JSON-like constructs in more detail in Sect. 5.3. The `RANDOM` sessions had a greater proportion of `CALL` events, likely as part of handling the randomly generated mouse events.

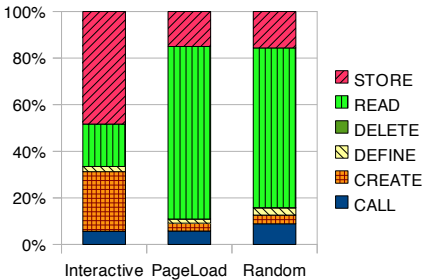


Fig. 6. Operation mix. Proportion of stores, reads, deletes, defines creates and calls performed by `eval`'d code.

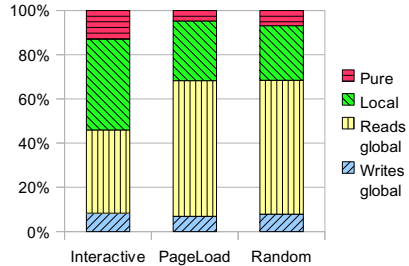


Fig. 7. Scope. Distribution of locality of operations performed by `eval`.

5.2 Scope

As with any JavaScript code, the code executed via `eval` may access both local and global variables. Code that does not access global state is self-contained and preferred. Our instrumentation determines statically what `eval` strings have no unbound variables and so are pure, and dynamically logs reads and writes to non-local variables. We categorize the locality of accesses within each call to `eval` into the following sets.

Pure	Access newly created objects.
Local	Access to local variables and objects.
Read Global	Same as Local + read properties of the global object.
Write Global	Same as Local + read/write/delete properties of the global object.

These categories help us understand the interplay between `eval` and encapsulation. The **Pure** category captures code that is restricted to creating objects and reading/writing their properties. All JSON code fits in this category. This is the safest category because it neither relies on nor affects the environment of the `eval`. The **Local** category includes cases where the `eval`'d code either reads, writes or deletes variables of the function that called the `eval` (or one of its lexically enclosing functions). The **Read Global** category extends the previous one with the ability to read properties of the global object. Potentially most dangerous is the **Write Global** category, consisting of `eval`'d code that also can add, modify or delete properties of the global object. For instance, writing to an undeclared variable will add and/or modify this variable in the global namespace. When the variable window is only defined in the global scope, then using this name for property access renders the side-effect evident. Also, in cases when the global object is aliased, resulting global read/writes may be underreported.

We found that `evals` in the **Pure** and **Writes Global** categories were scarce, while the other two categories were much more common. Fig. 7 shows the scope of operations performed by `evals` collected in each data set. While the number of **Pure** strings is quite low, the vast majority of `evals` are actually quite local: only 7 to 8% of all `evals` modify the global scope for all data sets. However, reads are more evenly split, 38 to 61% of all `evals` read from the global scope. It is reasonable to assume that many `eval` strings even in the **Local** and **Reads Global** categories have no side-effects outside the local scope, but are not self-contained, as their behavior will nonetheless depend on the global scope and if it were for using global functions only. Code passed to `eval` that is neither pure nor global (and so must be designed to work with a particular scope and `eval` call site) accounts for more than 41% of all `eval` strings in all data sets.

5.3 Patterns

There are many common patterns in the use of `eval`. Some are industry best practices, such as JSON, and asynchronous content and library loading. Others result from poor understanding of the language, repetition of old mistakes, or adapting to browser bugs. While it is not possible to be exhaustive, we have nevertheless identified 9 frequently occurring patterns of `eval` strings which can be detected by a simple syntactic check (a more precise description of how strings are categorized appears in Appendix A):

JSON	A JSON string or variant.
JSONP	A padded JSON string.
Library	One or more function definitions.
Read	Read access to an object's property.
Assign	Assignment to a local variable or object property.
Typeof	Type test expression.
Try	Trivial try/catch block.
Call	Simple function/method call.
Empty	Empty or blank string.
Other	Uncategorized string.

JSON-like constructs. Deserializing JSON is often seen as an acceptable use of `eval`. The **JSON** category covers strings that are in JSON syntax [6], as well as relaxed notions that permit equivalent JavaScript literals. The **JSONP** (JSON with padding) category covers strings which either assign a JSON expression to a variable or pass a JSON expression as an argument to a function. This pattern is often used for load balancing requests across domains. These other domain names violate the browser's same origin policy, precluding the use of XMLHttpRequest to load JSON from these servers. As a workaround, many standard libraries dynamically create JSONP expressions, typically a function call that takes a JSON argument. The function is a callback function that assigns the JSON data to a variable and processes that data.

Library loading. Libraries can be loaded by `<script>` tags in the document, but downloading, parsing, and evaluating scripts is synchronous with layout and other events. Blocking ensures deterministic page loading, since scripts can modify the DOM in-place. Although HTML5 introduces new mechanisms for deferred loading, their use is not widespread. A common workaround is to download the script asynchronously with AJAX, then execute it with `eval` at some later time. This does not block page parsing or rendering immediately, but leaves the programmer the burden of ensuring a known, consistent execution state. The **Library** category attempts to capture this pattern of use. A simple heuristic detects libraries: any `eval` string that is longer than 512 bytes and defines at least one function. Manual inspection revealed this to be a reasonable heuristic.

Field access. Access to properties of an object and to local variables is covered by the **Read** category. In the vast majority of situations, property reads can be replaced either by using JavaScript's hashmap access or by explicitly referencing the global scope. For instance, `eval('"foo."+x)` can be replaced by `foo[x]`. Concatenations like these are usually simple and repetitive. This pattern also often underlies a misunderstanding of arrays, such as using `eval('"subPointArr_'+i)` instead of making `subPointArr` an array. Another common use of `eval` is variable access. One reason why `evaling` might be useful comes from the scoping rules for `eval`. Using an aliased `eval` guarantees that accesses to variables will occur in the global scope. As mentioned before, this feature found little use. The **Assign** category comprises all statements that assign a value to a variable. A few sites have been found to use variable declarations within an `eval`. This actually modifies the local scope, and can alter the binding of variables around the `eval`.

Strange patterns. A strange expression pattern is the category which we call **Typeof** and which covers `typeof` expressions. For instance, `typeof(x)!="undefined"`. It in not

necessary to use `eval` for this expression. `typeof` is often used to check whether a variable is defined and define it if not, `if(typeof(x)===“undefined”) x={}`. However, in most cases, this too has clearer alternatives which use JavaScript’s hashmap style of field access. For instance, checking for the existence of a global variable can be done more clearly with `if(“x” in window)`. This misunderstanding can also be combined with a misunderstanding of object access, such as `eval(“typeof(zflag_+y0[+])!=“undefined”)` instead of making `zflag` a hashmap and using `y0[i]` in `zflags`.

Another case for which we have no satisfying explanation, labeled **Try**, is to `eval` `try/catch` blocks. For instance, `bbc.co.uk` `evals` `try{throw v=4}catch(e){}` which is semantically equivalent to `v=4` since the `throw` and the `catch` parts cancel each other out. Since it’s hard to imagine any reason to do this, we can only assume that this code is a strange corner-case of a code generator.

Function invocation. The **Call** category covers `evals` that invoke methods with parameters that are not padded JSON. A common case in this category is `document.getElementById`, the utility of which is particularly unclear since the parameter to `document.getElementById` is a string. If only the string parameter varies, then this can be done without `eval`. If the function called varies, `eval` can usually be avoided with hashmap syntax as described above. These are usually short and simple, such as `document.getElementById(“topadsblk01menu”)` and `update(obj)`. The latter could be done without `eval` using hashmap style access for the function name, for example `window[“update”](obj)`.

Other categories. The **Empty** category is made up of empty strings and strings containing only whitespaces. This pattern seems to be the default (empty) case for generated `eval` strings. Finally, the **Other** category captures any `eval’d` string not falling into the previous categories. In particular, it contains method calls interleaved with field access, like `foo.bar().zip`, but also more complex pieces of code that we did not categorize as a library. As an example consider the following code:

```
eval(“img1.src=’http://c.statcounter.com/t.php?ip_address=xx’;”);
```

which encodes data into a URL and sends an HTTP GET request in order to circumvent the same origin policy imposed by the DOM. It is also unclear why this example was passed to `eval`; we speculate that the particular mechanism of circumventing the same-origin policy is determined dynamically and the appropriate one used.

Distribution of categories. Almost all `eval` categories are present in each data set. Fig. 8 shows the number of *web sites* using each of the `eval` categories. The prevalence of **Other** `evals` is high, with 53 sites in INTERACTIVE using uncategorizable `evals`, 1020 sites in PAGeload and 1215 in RANDOM. Manual inspection suggests that there is no unifying category for these, and the actions performed are in fact quite diverse. Fig. 9 shows the number of *eval strings* in each category. Although uncategorizable `evals` are used in many sites, we have been able to categorize most strings, with 82%, 71% and 67% of strings categorized for INTERACTIVE, PAGeload and RANDOM, respectively. We see that loading libraries is common, and between 9% (for PAGeload) and 22% (for INTERACTIVE) of sites were detected doing so. Fig. 9 indicates that our method of categorizing libraries is accurate, as the number of actual `evals` in this category is quite low, at 2% for all data sets. Since most sites load only a few libraries, we expect the total number of `eval strings` in this category to be low.

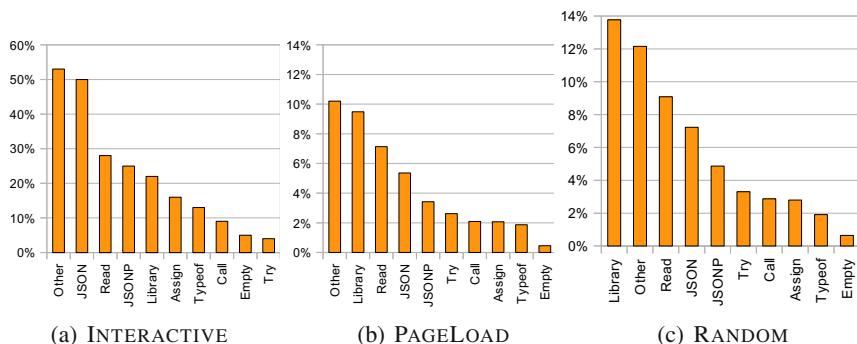


Fig. 8. Patterns by websites. Number of web sites in each data set with at least one eval argument in each category (a single web site can appear in multiple categories).

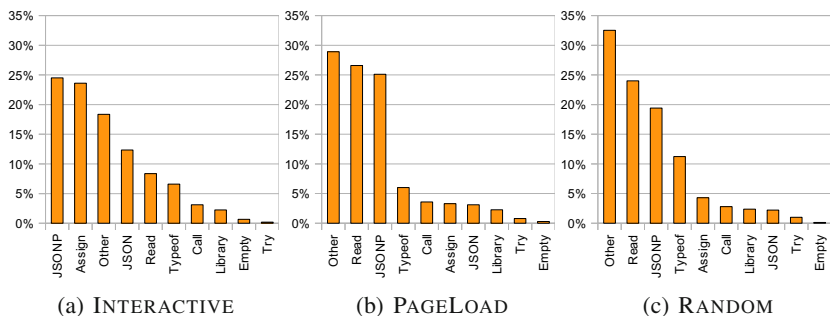


Fig. 9. Patterns. Ratio of evals in each category.

Both JSON and JSONP are quite common. In each data set, JSONP is at worst the third most common category in both Fig. 8 and Fig. 9, and JSON and JSONP strings accounted for between 22% (RANDOM) and 37% (INTERACTIVE) of all strings eval'd. Since most call sites do not change categories (discussed later in Section 5.5) these numbers indicate that analyses could make optimistic assumptions about the use of eval for JSON, but will need to accommodate the common pattern of JSON being assigned to a single, often easily-determinable, variable.

Most of the remaining evals are in the categories of simple accesses. Property and variable accesses, both simple accesses which generally have no side-effects, are in all data sets amongst the second to fifth most common categories for sites to use. They account for 8%, 27% and 24% of eval calls in INTERACTIVE, PAGEDLOAD and RANDOM, respectively. The most problematic categories⁷ appear in fewer sites, but seem to be used frequently in those sites where they do appear. However, this does not include uncategorized evals, which also have problematic and unpredictable behavior.

⁷ By problematic categories, we include evals with complex side effects such as assignments and declarations, and those categories with unconstrained behavior such as calls.

Impact on analysis. Most `eval` call sites in categories other than **Library**, **Other** and **Call** are replaceable by less dynamic features such as `JSON.parse`, hashmap access, and proper use of JavaScript arrays. On **INTERACTIVE**, these categories account for 76% of all `eval`'d strings; thus, a majority of `eval` uses are not necessary. Upon further investigation into instances of these categories, we believe that they are sufficiently simple to be replaced automatically. While we were able to confirm that best practices of JSON and asynchronous library loading are common uses of `eval`, other uses cannot be discounted: they are far from uncommon, and the sites that use them tend to use them quite often, and to perform diverse actions.

5.4 Provenance

Cross-site scripting attacks (XSS) often make use of `eval` to run arbitrary JavaScript code. To better understand where `eval`'d strings come from, we tagged all strings with provenance (tainting) information and instrumented all built-in string operations to preserve provenance information. The return values of certain HTML-specific operations (see below) were also tagged with provenance. We group strings by provenance in the following categories, where later categories may include all previous:

Constant	Strings that appear in the source code.
Composite	String constructed by concatenating constants and primitive values.
Synthetic	Strings that are constants in a nested <code>eval</code> .
DOM	Strings obtained from DOM or native calls.
AJAX	Strings that contain data retrieved from an AJAX call.
Cookies	Strings retrieved from a cookie or other persistent storage.
Input	Strings entered by a user into form elements.

For an example of **Synthetic** strings, consider `x=eval(""+document.location.href+""); y=eval(x)`. The argument to the first `eval` is from the DOM. However, because the first `eval` string is in fact a string literal, `x` is a string. `x` has **Synthetic** provenance, to distinguish it from string literals appearing in non-`eval` code (which have **Constant** provenance).

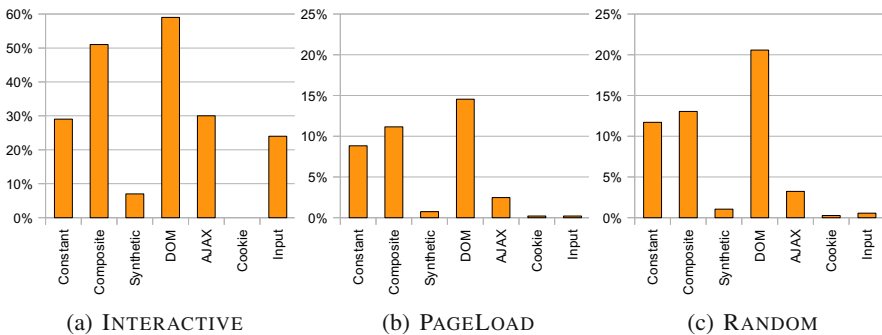


Fig. 10. Provenance by websites. Percentage of web sites using a string of given provenance at least once in an `eval` expression.

The **Constant** category includes string literals, and the **Composite** category includes strings created by concatenating string literals. The **DOM** category includes the result of DOM queries such as `document.body.innerHTML`) as well as native methods like `Date.toLocaleString()`.

The INTERACTIVE data set had a much higher appearance rate for all provenance types, which is not surprising. Fig. 10 shows the number of sites that pass strings of a given provenance to eval for our 3 data sets. The percentages of the PAGELOAD and RANDOM sets differ only slightly, and both had fewer strings of **AJAX** provenance.

Provenance data tells a more interesting story when aggregated by the provenance of each call to eval; Fig. 11 presents this view. For the INTERACTIVE data set, the dominant provenance of strings was **Composite**. More than 3,000 strings were constructed from composites of only constants and around 600 strings were just a constant in the source. The distribution of provenance is significantly different for the PAGELOAD and RANDOM data sets. For these, **DOM** and **Constant** are used in equal proportion, while **AJAX** is virtually nonexistent.

Provenance vs. Patterns. The eval pattern categories from Section 5.3 help to explain some of the surprising provenance data. Fig. 12 relates the patterns we found with provenance information. We had expected most JSON to originate from **AJAX**, as this is the standard way of dynamically loading data from a server. However, the **DOM** provenance outnumbers all others. The same holds for **Empty** and **Library** patterns. Upon further investigation into the low proportion of **AJAX** provenance, we found that,

for example, `google.com` retrieves most of its JSON as constant values by means of a dynamically-created `<script>` tag. This script contains code of the form `f({"x":3})`, where the parameter is a string containing a JSON object. However, instead of using the JSON string directly as a parameter (`f({"x":3})`), they parse the string in the function `f` using `eval`. Our provenance tracking will categorize this string as a compile time constant, as it is a constant in the dynamically created script tag. Because `google.com` stores its JavaScript on a separate subdomain, this convoluted pattern is necessary to circumvent the same-origin policy (under which the straightforward **AJAX** approach would be forbidden). Many major web sites have a similar separation of content.

In general, the simpler eval string patterns come from **Constant** and **Composite** sources. Looking at `Empty`, `Typeof`, `Read`, `Call`, `Assign` and `Try` as a group, 85% of these eval'd strings are constant or composite in **RANDOM**, with similar proportions in the other data sets. Many of these are often misused as replacements for arrays or hashmap syntax, so it is unsurprising that they are generated from constant strings.

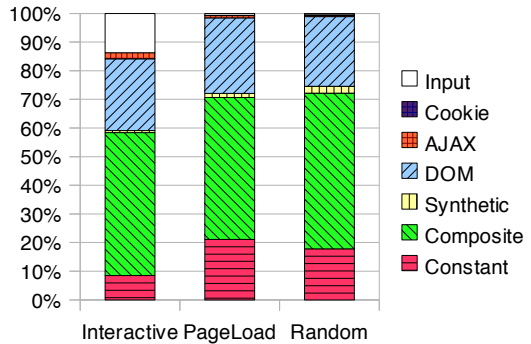
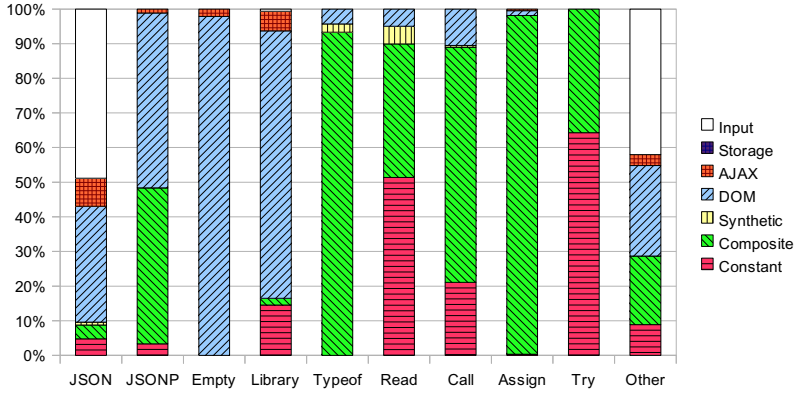
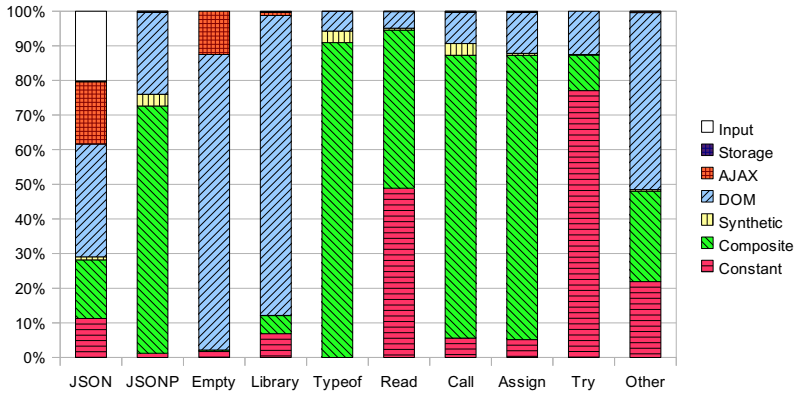


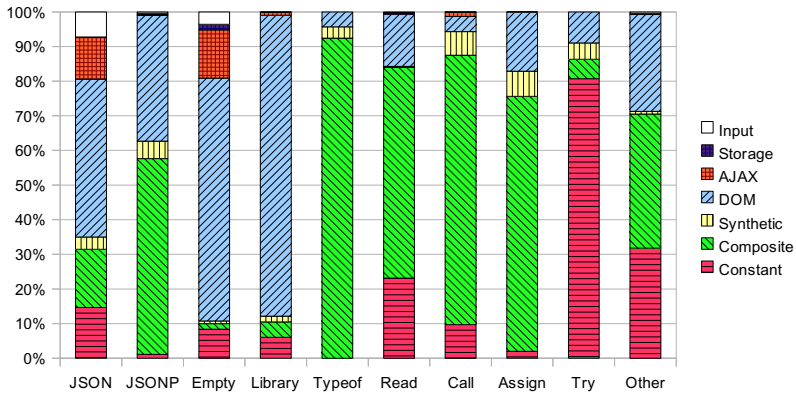
Fig. 11. Provenance. Proportion of strings with given provenance in eval'd strings for the three data sets.



(a) INTERACTIVE



(b) PAGELOAD



(c) RANDOM

Fig. 12. Provenance by Pattern. Distribution of string provenances across eval categories in each data set. X axis is the pattern that string falls into, Y axis is proportion of provenance in that category.

5.5 Consistency

Each eval call site is quite consistent with respect to the pattern of the string argument, but there are exceptions. Across all of our data sets, we observed only 399 eval call sites (1.4% of all call sites) with strings in multiple pattern categories, see Fig. 13. Many of these “polymorphic” cases were clearly a single centralized eval used from many branches and for many purposes. For instance, the following three strings are all eval’d by the same call site, found at

Patterns	1	2	3	4	5
Callsites	27553	303	92	3	1

Fig. 13. Consistency. Number of different patterns per call site.

```

window.location
dw_Inf.get(dw_Inf.ar)
dw_Inf.x0();

```

www.netcarshow.com in RANDOM (although the library that this eval belongs to is found at a few other sites as well). More perplexing call sites include ones that evals the strings “4”, “5” and “a”, callsites that alternate between simple constants and bound variables, and a call site that at times evaluated “(null)” (which happens be valid JSON) and at other times evaluated “(undefined)” (which is not). Another call site evals JSON strings in most cases, but sometimes evaluates JSON-like object literals which include function literals, which neither JSON nor relaxed JSON accept. Of the 399 eval call sites with strings in multiple patterns, the maximum number of categories was 5, at the call site mentioned above.

6 Other Faces of Eval

Eval is only one of several entry points to generate executable JavaScript code dynamically. This section reports on the use of the other methods of dynamic code generation available to programmers. We identified the following eight mechanisms of dynamic code generation provided to web programmers:

Eval	Call to eval, executing in local scope.
GlobalEval	Call to an alias executing in global scope.
Function	Create a new function from a pair of strings. (Global scope)
SetInterval	Execute a string periodically. (Global scope)
SetTimeout	Execute a string after a specified point in time. (Global scope)
ScriptCont	DOM operation that changes the contents of a script tag. (Global scope)
ScriptScr	DOM operation that changes the SRC attribute of a script tag. (Global scope)
Write	DOM operation that writes to the document in place. (Global scope)

The first three mechanisms are part of the JavaScript language. An example is the code `var y=Function("x", "print(x)")` which creates a new function that takes the parameter `x` and passes it to the `print` function. The following two mechanisms are not standardized but commonly implemented as properties of the window object. A simple example is `setTimeout("callback()",1000)` which invokes the `callback` function after 1 second. The final three mechanisms are related to DOM⁸ manipulation. **ScriptCont**

⁸ The Document Object Model (DOM) represents an HTML page as a tree, where nested tags are encoded as child nodes.

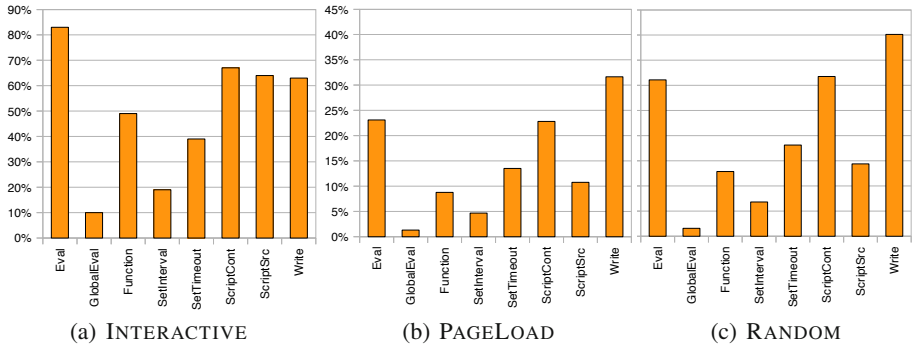


Fig. 14. Dynamic Code Generation by websites. Number of web sites in each data set that are dynamically creating scripts. The x-axis displays the mechanism to create the script, a single web site can appear multiple times.

covers changes to script tags such as setting the `text` or `innerHTML` property, or calling `appendChild(createTextNode(src))`, all of which change the source in the script tag. The **ScriptSrc** covers modifications of the `src` attribute of a script tag, which downloads the given resource and executes its code. The **Write** category covers uses of `document.write` to manipulate the DOM in-place while executing JavaScript code. (Libraries that contain `document.write` cannot be loaded asynchronously.) The use of write is discouraged. Consider the the example below, where the first line outputs “<scr” to the document which is concatenated in place with “ipt>” to create a script tag:

```
<script>document.write("<scr");</script>
ipt>alert("this is malicious");</script>
```

The above code is typical of an attack that tries to fool malware filters.

The prevalence of the different mechanisms varies widely among the data sets, especially between the interactively and automatically-gathered data sets. Fig. 14 displays how many sites use each mechanism at least once. In the INTERACTIVE data set, **Eval** is predominant (present in 83% of sites), but in PAGELOAD and RANDOM this mechanism was only used in 23% and 31% of sites, respectively. The use of the global eval variant (**GlobalEval**) is minor (10% of the pages in the INTERACTIVE data set) and even less so in the other data sets (1.3% and 1.6% for PAGELOAD and RANDOM). The **Function** constructor is frequently used by sites in the INTERACTIVE data set (49%), while the other two data sets make only limited use of it (between 8.8% and 13%).

The remaining non-JavaScript mechanisms are used widely. **SetTimeout** is generally used by twice as many sites as **SetInterval**. **SetTimeout** appears in 39% of the sites in INTERACTIVE data set, and for the other data sets between 13% and 18%. Setting the content of a script tag is widespread in the INTERACTIVE data, where 67% of the sites use it, compared to only 23% in the PAGELOAD and 32% in the RANDOM data. Setting the `src` attribute of a script tag is only widespread in the INTERACTIVE (at 64%) data set, compared to 10–15% in the other data sets. This seems to be a result of the most popular sites using this mechanism to load content from servers on a different domain. Writing script tags to the DOM is popular for all data sets, with 64% of the INTERACTIVE sites doing this, 32% of PAGELOAD and 40% of RANDOM.

When counted by number of actual uses, the **Eval** construct constitutes a clear *minority*; this is worrying, since other code generation mechanisms tend to be overlooked or ignored in the literature. Fig. 15 shows the distribution of the different mechanisms to dynamically create code. For the INTERACTIVE data set, the **Function** constructor was the most commonly used mechanism, despite **Eval** being present in many more sites. Usage of **SetTimeout** is also quite frequent, accounting for more invocations than **Write**, **ScriptSrc**, and **ScriptCont** combined, despite appearing in fewer sites than those mechanisms. This pattern makes sense when one considers that uses of **SetTimeout** frequently recur (in lieu of using **SetInterval**). For the PAGELOAD data set it is interesting to note that **SetTimeout** is used most frequently, **SetInterval** is rarely used, and 7% of scripts written directly to the DOM. This distribution corresponds well with initial setup of the web page, where some tasks are deferred by **SetTimeout**. This is reinforced by the distribution of the RANDOM data. It creates more scripts by means of **Eval**, and is the only data set where **SetInterval** plays a significant role for script creation. We attribute this to the greater dynamism triggered by our random clicking strategy.

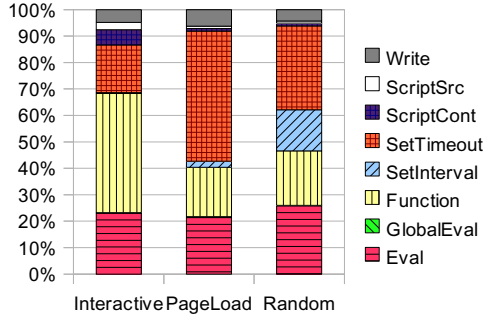


Fig. 15. Dynamic Code Generation. Distribution of mechanisms used to dynamically create scripts per data set.

Classifying the behavior of code created by each of the mechanisms according to the patterns in Sect. 5.3 gives an even better picture of how these mechanisms are commonly used. This classification is depicted in Fig. 16. For all data sets, the local and global **Eval** is used to load the most diverse code, with about 9 significant patterns for these two mechanisms. All the other mechanisms are far less diverse, falling into 3–7 of our defined patterns. **setInterval** and **setTimeout** in particular are used almost exclusively with simple function calls (bearing in mind that by JSONP’s definition, it is

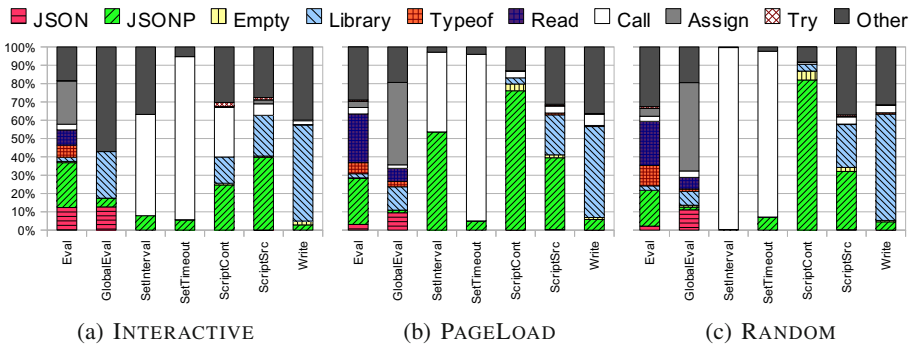


Fig. 16. Patterns by Dynamic Code Generation. Distribution of code patterns per mechanism of dynamically creating scripts.

often also a simple function call). This consistency suggests that these functions could be speculatively optimized or replaced by safer alternatives in the JavaScript runtime. Even for the other less predictable mechanisms, there is a sufficient lack of diversity that an optimizing compiler could provide faster or safer alternatives according to our patterns.

7 Case Studies

We will now look at individual websites and give examples of their use of `eval`.

Heute.de: The German news site *heute.de* (from our RANDOM data set) has a representative example of the naïve use of `eval` found in many sites, in this case in a snippet which is also found on several other sites. The website contains 49,174 bytes of JavaScript code, with a paltry 136 bytes of `eval` in 9 calls from the same call site. The `eval`-using code is summarized in the following snippet:

```
var flashVersion = parse();
flash2Installed = flashVersion == 2;
flash3Installed = flashVersion == 3;
... // same for 4 to 10
flash11Installed = flashVersion == 11;
for (var i = 2; i <= maxVersion; i++)
  if (eval("flash"+i+"Installed")==true)
    actualVersion = i;
```

This example is enlightening in its utter disregard for any consideration of style, legibility and performance. The purpose of the code is to set global variable `actualVersion` to the version of Flash plugin available in the browser. This is achieved by, first, storing the version number in the local variable `flashVersion`, then creating 10 new global variables `flashiInstalled`

(which pollute the namespace and are never used again). Then, to save space perhaps, a loop iterates over an `eval` that reads a constructed variable name and sets `actualVersion`. (As an aside, the loop guard, `maxVersion`, is 10, thus `flash11Installed` will never be seen.) In this case there is no reason to use `eval` at all, the entire code snippet could be replaced by the more direct one-liner: `actualVersion = parse()`.

Trainenquiry.com: This Indian train schedule site (also from RANDOM) has 42,135 bytes of JavaScript code and 163 bytes of `eval` strings across three call sites, all in the `ValidatorHookupEvent` function (irrelevant code elided):

```
function ValidatorHookupEvent(control, eventType, functionPrefix) {
  var ev;
  eval("ev = control." + eventType + ";");
  eval("control." + eventType + " = func;");
  if (typeof(val.evalfunction) == "string")
    eval("val.evalfunction = " + val.evalfunction + ";");
```

The first two cases are simple misunderstanding of JavaScript which could be expressed more efficiently and succinctly as hash map accesses:

```
ev = control[eventType];
control[eventType] = func;
```

The last one is worth explaining in a little bit more detail. The property `val.evalfunction` may hold a string, in which case, it is taken to be the name of the function that should

be stored in that property. The conditional will use `eval` to replace the string with a reference to a function object. This could also have been expressed as

```
if (typeof(val.evalfunction) == "string")
    val.evalfunction = window[val.evalfunction];
```

where `eval` is again replaced with hash map access to a global property.

Ask.com: Because it loads functionality from several different sources, several of which use `eval`-equivalent behavior, and it contains a wide variety of behaviors generated by dynamic code, `ask.com` is an interesting case study. This site loads 2.22MB unique code, 1.39MB of code passed to all variants of `eval`, and 3.77KB passed to 409 `eval` calls originating from 48 callsites. The code passed to variants of `eval` consists of several large libraries from different sources (through `<script>` tag generation), and two of them, as well as the host, also dynamically generate code. We have excerpted several examples. The site contains ads, and one ad agency's scripts are loaded dynamically by adding `<script>` tags to the document by means of `document.write`.

```
document.write("<scr"+"ipt type='text/javascript'
src='http://afe.specificclick.net/?l=12915&sz=300x250&wr=j&t=j&u="+u
+"&r="+r+"&rnd="+sm_random+"'></scr"+"ipt">");
```

The behavior of the script added is to save some tracking data, then dynamically load more scripts which themselves load more scripts. This is done by setting the `src` attribute of a script tag and using `document.write`.

```
var _comscore = _comscore || []; _comscore.push({ c1: "8", c2: "2101", ... });
(function() { var s = document.createElement("script"), ...; s.async = true; ...;
document.write("<SCRI"+"PT src='http://ads....'"></SCRI"+"PT">");
```

As a search query is entered, `ask.com` attempts to auto-complete it. Because auto-completion is performed by a server on a different domain, `XMLHttpRequest` is not an option and instead a `<script>` tag is created with the request encoded into the URL. The script loaded in response is a JSONP string. Given the limited portable options for cross-domain communication, this is reasonable.

```
searchSuggestion(["who",["<span ...>who</span> is justin bieber",...]]);
```

An initialization routine is deferred by means of `setTimeout` with a string argument, presumably to assure that it does not interfere with the loading of the remaining source.

```
setTimeout("JASK.currentTime.init()",JASK.currentTime.SECOND);
```

Since this string is a constant, it could be replaced with a function. We intercepted four different ways to initialize a local variable coming from the same `eval` call site:

```
function(str){...; eval("var p="+str+"); return p;}
```

This attempt at JSON deserialization suffers from the dual misconceptions that `eval` can only evaluate statements and not expressions, and that `eval` is the only way to deserialize JSON. The `eval` can be replaced portably by:

```
if("JSON" in window) return JSON.parse(str); else return eval("(" + str + ")");
```

The reCAPTCHA library updates state in a way that is similar to JSONP, but performs both an assignment and a call, and also uses a relaxed form of JSON. It is loaded similarly to the auto-completion example above.

```
var RecaptchaState = {... timeout : 18000}; Recaptcha.challenge_callback();
```

The following line, which assigns a value to itself, intuitively makes no sense.

```
RichMediaCreative_1298 = RichMediaCreative_1298;
```

This odd behavior is clarified by the original code. A function is loaded with a name containing a unique ID (a timestamp, in fact), and used from other loaded code under that name. Presumably for fear of a miscommunication, `eval` is used to assure that the created function is assigned to the name that the other code expects.

```
eval("RichMediaCreative_"+plcrInfo_1298.uniqueId+"=RichMediaCreative_1298;");
```

Since the function exists in the global scope, this case is easily replaceable by `hashmap` syntax over the `window` object.

8 Related Work

Empirical data on real-world usage of language features is generally missing or limited to a small corpus. In previous work, we investigated the dynamic behavior of real-world JavaScript applications [18]. That result, on a corpus of 103 web sites, confirmed that `eval` is widely used for a variety of purposes, but in that effort we did not scale up the analysis to a larger corpus or provide a detailed analysis of `eval` itself. Ratana-worabhan *et al.* have performed a similar study of JavaScript behavior [17] focusing on performance and memory behavior. There have been studies of JavaScript's dynamic behavior as it applies to security [21,7] including the role of `eval`, but the behaviors studied were restricted to security properties. Holkner and Harland [10] conducted a study of dynamic features in Python, which includes a discussion of `eval`. Their study concluded that there is a clear phase distinction in Python programs. In their corpus dynamic features occur mostly at initialization and less so during the main computation. Their study detected some uses of `eval`, but their corpus was relatively small so they could not generalize their observations about uses of `eval`. Other languages have facilities similar to `eval`. Livshits *et al.* did static analysis of Java reflection in [14], and Christensen *et al.* [3] analyze the reflection behavior of Java programs to improve analysis precision for their analysis of string expressions.

9 Conclusion

This paper has provided the first large-scale study of the runtime behavior of JavaScript's `eval` function. Our study, based on a corpus of the 10,000 most popular websites on the Internet, captures common practices and patterns of web programming. We used an instrumented web browser to gather execution traces, string provenance information, and string inputs to `eval`. A number of lessons can be drawn from our study. First and foremost, we confirm that `eval` usage is pervasive. We observed that between 50%

and 82% of the most popular websites used `eval`. Clearly, `eval` is not necessarily evil. Loading scripts or data asynchronously is considered a best practice for backwards-compatibility and browser performance, because there is no other way to do this. While JSON is common, we found that `eval` is not used solely for JSON deserialization. Even if we allowed relaxed JSON and JSONP notation, this accounts for at most 37% of all calls. Thus, nearly two thirds of the calls do in fact use other language features. It seems that `eval` is indeed an often misused feature. While many uses `eval` were legitimate, many were unnecessary and could be replaced with equivalent and safer code.

We started this work with the hope that it would show that `eval` can be replaced by other features. Unfortunately our data does not support this conclusion. Removing `eval` from the language is not in and of itself a solution; `eval` is a convenient way of providing a range of features that weren't planned for by the language designers. For example, JSON was created to support (de-)serialization of JavaScript objects. It was straightforward to implement with `eval`, and it is now supported directly in ECMAScript 5. Standards for safer and more consistent library loading have been proposed, e.g. as part of CommonJS. Most accepted uses of `eval` have been transformed into libraries or new language features recently, and as such no best practices recommends usage of `eval`. However it is still needed for some use cases such as code generation, which either have not or can not be encapsulated into safer strategies. On the positive side, our categorization was extremely simple, and yet covered the vast majority of `eval` strings. The categories were chosen to be as restrictive as they are to assure that they are easily replaced by other mechanisms. Restricting ourselves to `eval`'s in which all named variables refer to the global scope, many patterns can be replaced by more disciplined code. The following table illustrates some simple replacements for our patterns.

JSON	<code>JSON.parse(str)</code>
JSONP	<code>window[id] = JSON.parse(str)</code> or <code>window[id](JSON.parse(str))</code>
Read	<code>window[id]</code> or <code>window[id][propertyName]</code>
Assign	<code>window[id] = window[id]</code> or <code>window[id][propertyName]=window[id]</code>
Typeof	<code>typeof(window[id])</code> or <code>id in window</code>
Try	(Not trivially replaceable)
Call	<code>window[id](window[id], ...)</code> or <code>window[id].apply(window, [...])</code>
Empty	<code>undefined</code> or <code>void 0</code>

Furthermore, more than two thirds of the `eval` strings in these categories listed above are of constant or composite provenance (66.3%, 81.9% and 75.1% in INTERACTIVE, PAGeload and RANDOM, respectively) giving a limited number of possible names to be referred to. All but one of these replacements depend on JavaScript's hashmap syntax, which can be used to access properties of objects by string name, but not variables in scope. Since the global scope is also exposed as an object, `window`, this is sufficient for accessing variables which happen to be in the global scope. However, at least a quarter to a half of `eval` strings refer to local variables (locality "local": 41.1%, 27.0% and 24.7% in INTERACTIVE, PAGeload and RANDOM, respectively; likely everything but "pure"), possibly precluding the use of hashmap syntax. Many of these can be replaced somewhat less trivially in existing code by putting variables which would be accessed by a dynamic name into an object and using hashmap syntax, but for the general case an

extension to JavaScript which would allow to access local variables dynamically would greatly reduce the need for `eval`.

Acknowledgments. We thank Sylvain Lebesne and Keegan Hernandez for their work on the tracing framework and data recording; as well as Andreas Gal, Shriram Krishnamurthi, Ben Livshits, Peter Thiemann, and Ben Zorn for inspiration and comments. This work was partially supported by a grant from Microsoft Research and by the ONR award N000140910754.

References

1. Anderson, C., Drossopoulou, S.: BabyJ: From Object Based to Class Based Programming via Types. *Electr. Notes in Theor. Comput. Sci.* 82(7), 53–81 (2003)
2. Anderson, C., Giannini, P.: Type Checking for JavaScript. *Electr. Notes Theor. Comput. Sci.* 138(2), 37–58 (2005)
3. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) *SAS 2003*. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
4. Chugh, R., Meister, J.A., Jhala, R., Lerner, S.: Staged Information Flow for JavaScript. In: *Conference on Programming Language Design and Implementation (PLDI)*, pp. 50–62 (2009)
5. Egele, M., Wurzinger, P., Kruegel, C., Kirda, E.: Defending Browsers against Drive-by Downloads: Mitigating Heap-Spraying Code Injection Attacks. In: Flegel, U., Bruschi, D. (eds.) *DIMVA 2009*. LNCS, vol. 5587, pp. 88–106. Springer, Heidelberg (2009)
6. European Association for Standardizing Information and Communication Systems (ECMA): *ECMA-262: ECMAScript Language Specification*. 5th edn. (December 2009)
7. Feinstein, B., Peck, D.: Caffeine Monkey: Automated Collection, Detection and Analysis of Malicious JavaScript. In: *Black Hat USA 2007* (2007)
8. Guarnieri, S., Livshits, B.: Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In: *USENIX Security Symposium*, pp. 151–197 (2009)
9. Guha, A., Krishnamurthi, S., Jim, T.: Using Static Analysis for Ajax Intrusion Detection. In: *Conference on World Wide Web (WWW)*, pp. 561–570 (2009)
10. Holkner, A., Harland, J.: Evaluating the Dynamic Behaviour of Python Applications. In: *Proceedings of the Thirty-Second Australasian Conference on Computer Science, ACSC 2009*, vol. 91, pp. 19–28. Australian Computer Society, Inc., Darlinghurst (2009)
11. Jang, D., Choe, K.M.: Points-to Analysis for JavaScript. In: *Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009*, pp. 1930–1937. ACM, New York (2009)
12. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications. In: *CCS 2010: Proceedings of the 17th ACM Conference on Computer and Communications Security*, pp. 270–283. ACM, New York (2010)
13. Jensen, S.H., Møller, A., Thiemann, P.: Type Analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) *SAS 2009*. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
14. Livshits, B., Whaley, J., Lam, M.S.: Reflection Analysis for Java. In: Yi, K. (ed.) *APLAS 2005*. LNCS, vol. 3780, pp. 139–160. Springer, Heidelberg (2005)
15. Maffeis, S., Mitchell, J.C., Taly, A.: Isolating JavaScript with Filters, Rewriting, and Wrappers. In: Backes, M., Ning, P. (eds.) *ESORICS 2009*. LNCS, vol. 5789, pp. 505–522. Springer, Heidelberg (2009)
16. McCarthy, J.: History of LISP. In: *History of programming languages (HOPL)* (1978)

17. Ratanaworabhan, P., Livshits, B., Zorn, B.: JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In: USENIX Conference on Web Application Development (WebApps) (June 2010)
18. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An Analysis of the Dynamic Behavior of JavaScript Programs. In: Programming Language Design and Implementation Conference, PLDI (2010)
19. Rieck, K., Krueger, T., Dewald, A.: Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In: Annual Computer Security Applications Conference, ACSAC (2010)
20. Thiemann, P.: Towards a Type System for Analyzing JavaScript Programs. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005)
21. Yue, C., Wang, H.: Characterizing Insecure JavaScript Practices on the Web. In: World Wide Web Conference, WWW (2009)

A Patterns

Patterns are determined in strings by a simple tool itself written in JavaScript, using `JSON.parse` and various regular expressions. Its algorithm is as follows: With `eval` string `str`:

- If `str` starts and ends with a (and), remove them. This is a common workaround to force certain engines to interpret the string as an expression instead of a statement.
- Strip whitespace from the beginning and end, and comments from any location in the string.
- If `JSON.parse(str)` does not throw an exception, return **JSON**.
- Relax `str` into `str_relaxed` (the relaxation procedure is explained below).
- If `JSON.parse(str_relaxed)` does not throw an exception, return **JSON**.
- Test `str` against regular expressions to determine other patterns.

The regular expressions (shown here in the order they are tested) are:

JSONP	<code>/[A-Za-z0-9_\$. \t\[\]]*=[\t\r\n]*(.*);*\$/ and matched substring 1 must be JSON or relaxed JSON</code>
Empty	<code>/\$/</code>
Library	<code>/function *[A-Za-z0-9_\$. \t\[\]]* *(/ and string must be greater than 512 bytes</code>
Typeof	<code>/typeof *(? *[A-Za-z0-9_\$. \t\[\]]*?)/ or /typeof *\([? *[A-Za-z0-9_\$. \t\[\]]*?)? *(!=<> +[\(\)]*\)\)? *?:? */</code>
Read	<code>/[A-Za-z0-9_\$. \t\[\]]*\$/ or /[A-Za-z0-9_\$. \t\[\]]*\$/</code>
Call	<code>/[A-Za-z0-9_\$. \t\[\]]*\([A-Za-z0-9_\$. \t\[\]]*\.[\r\n]*\);?\$/</code>
Assign	<code>/[A-Za-z0-9_\$. \t\[\]]* *=[A-Za-z0-9_\$. \t\[\]]* *?[\r\n]*\$/ or /var [A-Za-z0-9_\$. \t\[\]]* *(=[A-Za-z0-9_\$. \t\[\]]* *)?;?\$/</code>
Try	<code>/try *\{[\r\n]*\} *catch *\{[\r\n]*\} *\{[\r\n]*\} *;?\$/</code>

All other strings are categorized as Other.

The relaxation procedure is a simple process that replaces most JSON-like strings with strict JSON strings. Single-quoted strings are replaced with double-quoted strings (e.g. `{'foo':0}` becomes `{"foo":0}`), unquoted names are quoted (`{foo:0}` becomes `{"foo":0}`) and a form of string escapes not accepted by JSON (`\x`) are replaced by their JSON equivalent (`\u`).

B Performance Impact of Eval

The WebKit JavaScript engine will generate different bytecodes for local variable access when a function calls `eval`.

Consider the two functions in Fig. 17. Because of the presence of `eval`, the translation of function `E()` must do dynamic, by-name lookup of `x` (opcodes `resolve_with_base`, `put_by_id`, `resolve`), whereas `NoE()` simply refers to statically-known global offsets (opcodes `get_global_var`, `put_global_var`). This is a direct example of the potential impact of `eval` on performance as the code on the left will run slower in a WebKit.

```

function E() {
    eval(evalstr); x++;
    return x;
}
enter
init_jazy_reg r0
init_jazy_reg r2
init_jazy_reg r1
create_activation r0
resolve_with_base r4, r3,
                    eval(@id0)
resolve r5, evalstr(@id1)
call_eval r3, 2, 12
op_call_put_result r3
resolve_with_base r4, r3, x(@id2)
pre_inc r3
put_by_id r4, x(@id2), r3
resolve r3, x(@id2)
tear_off_activation r0, r2
ret r3

function NoE() {
    id(evalstr);
    x++;
    return x;
}
enter
get_global_var r0, -8
mov r1, undefined(@k0)
get_global_var r2, -12
call r0, 2, 9
get_global_var r0, -11
pre_inc r0
put_global_var -11, r0
get_global_var r0, -11
ret r0

```

Fig. 17. Bytecode generated by WebKit

C Local vs. Global Scope

The `eval` function provides two modi operandi. Called directly, it executes in the local scope and only variables that are not declared in that scope will bind to the local scope. However, if called through an alias, then `eval` executes in the global scopes and all variables, declared or undeclared in the `eval` string, bind to the global scope. In the following program the first `eval` executes in the local scope and thus assigns to the local variable `x`, while the call to the alias of `eval` assigns to the global variable `x`.

```

1 (function() { // the anonymous function creates its local scope
2   var x = eval("x = 4"); // assigns 4 to the local variable x twice
3   var e = eval; // alias eval to call it in the global scope
4   x = e("x = 4");})(); // first assigns 4 to the global variable x and then to the local variable

```

Using Structure-Based Recommendations to Facilitate Discoverability in APIs

Ekwa Duala-Ekoko and Martin P. Robillard

School of Computer Science, McGill University
Montréal, Québec, Canada
{ekwa,martin}@cs.mcgill.ca

Abstract. Empirical evidence indicates that developers face significant hurdles when the API elements necessary to implement a task are not accessible from the types they are working with. We propose an approach that leverages the structural relationships between API elements to make API methods or types not accessible from a given API type more discoverable. We implemented our approach as an extension to the content assist feature of the Eclipse IDE, in a tool called API Explorer. API Explorer facilitates discoverability in APIs by recommending methods or types, which although not directly reachable from the type a developer is currently working with, may be relevant to solving a programming task. In a case study evaluation, participants experienced little difficulty selecting relevant API elements from the recommendations made by API Explorer, and found the assistance provided by API Explorer helpful in surmounting discoverability hurdles in multiple tasks and various contexts. The results provide evidence that relevant API elements not accessible from the type a developer is working with could be efficiently located through guidance based on structural relationships.

1 Introduction

Application Programming Interfaces (APIs) play a central role in modern-day software development. Software developers often favor *reuse* of code libraries or frameworks through APIs over *re-invention* as reuse holds promise of increased productivity. Learning how to use APIs, however, presents several challenges to both novice and expert developers [4,12,15,16]. One such challenge, referred to as the *discoverability problem*, highlights the difficulty faced by a developer looking for the types and methods of an API necessary to implement a programming task [12,16]. Empirical evidence indicates that when working on a programming task, most developers look for a *main-type* central to the scenario to be implemented and explore an API by examining the methods and types referenced in the method signatures of the main-type [16]. As a result, a developer may be at a significant disadvantage when an API method essential to a task is located on a *helper-type* not directly accessible from the main-type, or when other essential types are not referenced in the signature of the methods on a main-type. For instance, Stylos et al. observed that placing a “send” method on a helper-type such as `EmailTransport.send(EmailMessage)`, instead of having it on the main-type such as `EmailMessage.send()`, significantly hinders the process of learning how to use APIs; they observed that developers were two to eleven times faster

```

Properties props = System.getProperties();
props.put("mail.smtp.host", "localhost");
Session ses = Session.getInstance(props);
Message message = new MimeMessage(ses);
//set other attributes of Message

Transport.send(message);

```

JavaMail API

```

SimpleEmail message = new SimpleEmail();
message.setHostName("localhost");
//set other attributes of email

message.send();

```

Apache Wrapper for JavaMail

Fig. 1. Apache Commons wrapper for the JavaMail API placed the “send” method on the main-type, and simplified the process of creating an email message object by providing a default constructor

at combining multiple objects when relevant API methods and types were accessible from the main-type [16]. A different study which looked at the usability tradeoff between the use of the Factory pattern or a constructor for object construction also reported that developers required significantly more time using a factory than a constructor because factory classes and methods are not easily discoverable from the main-type [4].

A potential solution to improving discoverability in APIs is to restructure an API to make the methods and types essential to the use of a main-type discoverable. For instance, moving the “send” method from `EmailTransport` to `EmailMessage` and providing constructors for object construction instead of the Factory pattern would improve discoverability. However, such a restructuring may not always be beneficial as it could negatively impact other desirable features of an API such as its performance and evolvability, and the client code may also become broken. A second solution to the discoverability problem is to provide an API wrapper. For instance, we are aware of over six API wrappers for the JavaMail¹ API, all aimed at simplifying the process of composing and delivering an email message. One such wrapper, provided by the Apache Commons project, underscores the discoverability issues with the JavaMail API by placing the “send” method on the main-type, and by simplifying object construction through the use of a constructor (see Figure 1). The use of API wrappers to resolve discoverability problems is promising but may be expensive, and introduces maintenance and versioning problems. Furthermore, developing wrappers introduces the risk of unwittingly altering the behavior of the original API.

In this paper, we propose a novel and an inexpensive approach for improving the discoverability of API elements. Our approach is based on the intuition that the structural relationships between API elements, such as method-parameter

¹ <http://java.sun.com/products/javamail>

relationships, return-type relationships and subtype relationships, can be leveraged to make discoverable the methods and types that are not directly accessible from a main-type. For instance, we can use the fact that `EmailTransport.send(EmailMessage)` takes `EmailMessage` as a parameter to recommend the “send” method of the `EmailTransport` class when a developer looks for a “send” method, or something similar, on the `EmailMessage` class. Similar recommendations can be made for object construction from factory methods, public methods, or subtypes. These can be accomplished without trading off other desirable API features to make elements discoverable, or the need to create and maintain API wrappers.

To investigate our intuition, we built a recommendation system, called *API Explorer*² which makes use of a special-purpose dependency graph for APIs to provide recommendations based on the structural context in which assistance is requested. We implemented API Explorer as a novel extension of the content assist feature of the Eclipse IDE. Content assist in Eclipse, or IntelliSense as it is called in Microsoft Visual Studio, is limited to showing only the methods available on the object on which it is invoked. API Explorer extends content assist with support for recommending relevant methods on other objects, locating API elements relevant to the use of a method or type, and also providing support for combining the recommended elements. We evaluated API Explorer through a multiple-case study in which eight participants were asked to complete the same four programming tasks using four different real-world APIs, each task presenting multiple discoverability challenges. The results of the study was consistent across the participants and the tasks, and show that API Explorer is effective in assisting a developer discover relevant helper-types not accessible from a main-type. The results also show that the use of structural relationships, combined with the use of content assist to generate and present recommendations, could be a viable, and an inexpensive, alternative when seeking to improve discoverability in APIs. We make the following contributions:

- We present an approach that uses the structural relationships between API elements to make discoverable helper-types not accessible from a main-type.
- We provide API Explorer, a publicly available plugin for Eclipse that embodies our approach. API Explorer is the first tool, to our knowledge, that can recommend relevant API methods on other objects through the content assist feature of an IDE.
- We present a detailed analysis of data from 32 programming sessions of participants using API Explorer with real-world APIs, showing how our approach is effective in helping developers discover helper-types not reachable from a main-type, and helping us understand the contexts in which the approach would not be effective.

We continue in the next section with an example scenario that highlights typical discoverability hurdles observed in previous studies on API usability.

² API Explorer is available at: www.cs.mcgill.ca/~swevo/explorer

2 Motivation

Consider a scenario in which a developer has to implement a solution to compose and deliver an email message using the JavaMail API. Going through the documentation of JavaMail, the developer found `Message`, the main-type representing an email message. The developer then proceeds by attempting to construct an object of type `Message` from its default constructor³ and encounters the first discoverability hurdle: `Message` is an abstract class. Creating an object of type `Message` requires three helper-types (`MimeMessage`, `Session`, and `Properties`), none of which are directly accessible from `Message` (i.e., these helper-types are not referenced or reachable from any of the public members of `Message`). Eventually, after spending some time going through the documentation, or code examples on the Web, the developer would locate all the types necessary to construct a `Message` object, and also the information on how these types should be combined. Once `Message` is created and all the necessary attributes are set, the developer then proceeds to send the email and encounters the second discoverability hurdle: there is no method on the `Message` object that provides the “send” functionality. The developer must therefore spend more time looking for a helper-type with a method that could be used to send the `Message` object. The code completion feature of the IDE is not helpful because it can only display the methods available on `Message` and provides no easy way to discover the existence of a helper-type with a send method. Also, the traditional search tools that come with IDEs do not provide direct support to locate multiple helper-types from a main-type. A developer would have to combine the results from multiple tools (e.g., the type hierarchy and reference search tools) and filter out the search results before potentially finding the relevant helper-types.

This scenario describes a conceptually simple task but highlights discoverability hurdles commonly faced by developers in practice: “...in real world APIs like Java’s JDK and Microsoft’s .NET, it frequently seems to be the case that the classes [helper-types] one needs are not referenced by the classes [main-type] with which one starts...” [16, p.2]. Thirteen out of twenty participants in a separate exploratory study we conducted to investigate the challenges developers encounter when learning to use APIs experienced some difficulty locating the helper-types relevant to implementing a programming task [2]. We observed that the participants relied on imperfect proxies such as domain knowledge or their expectation of how an API should be structured when looking for helper-types not referenced by a main-type. These attributes are often not consistent across different APIs, and may be misleading, resulting in unsuccessful searches and wasted efforts. This observation raises two research challenges:

- How can we assist developers in efficiently discovering helper-types not accessible from a main-type?
- How can we assist developers in the process of combining these related types to implement a task?

³ Three separate studies observed that most developers, both novice and experts alike, begin object construction by attempting to use the default constructor [4][15][16].

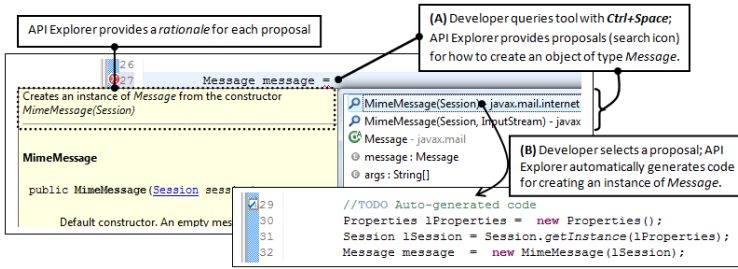


Fig. 2. API Explorer shows the developer the types required to construct an instance of *Message* and generates code which illustrate how to combine these types

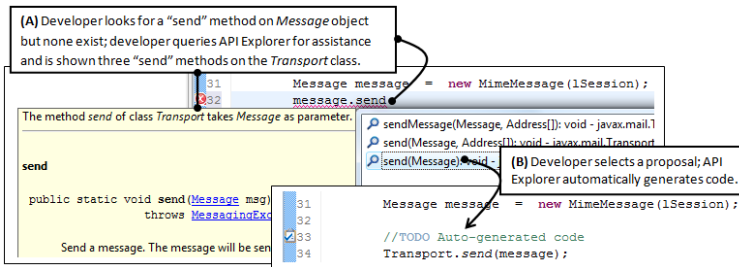


Fig. 3. API Explorer recommends three send methods on the *Transport* class which can be used for sending an email *Message* object

We hypothesize that structural relationships between API elements can be leveraged as beacons to assist developers locate helper-types not accessible from a main-type, and in combining these related types. In the next section, we discuss how API Explorer, a tool developed to investigate our hypothesis, could have assisted the developer quickly surmount the hurdles encountered above. We present the heuristics and algorithms enabling API Explorer in Section 4.

3 API Explorer

API Explorer generates recommendations that would assist a developer discover helper-types not accessible from a main-type based on the structural context in which help is requested.

Faced with an object construction hurdle, a developer would query API Explorer for assistance by invoking content assist after the assignment operator. For instance, in the example above, the developer would enter *Message m =*, then the key sequence *Ctrl+Space*, and API Explorer would instantly display two options for creating a *Message* object from *MimeMessage* (see Figure 2(A)). API Explorer can provide assistance for creating objects from constructors, subtypes, factory methods, public methods, or static methods. Selecting a recommendation reveals a *hoverdoc*, containing a *rationale*, that explains why the element

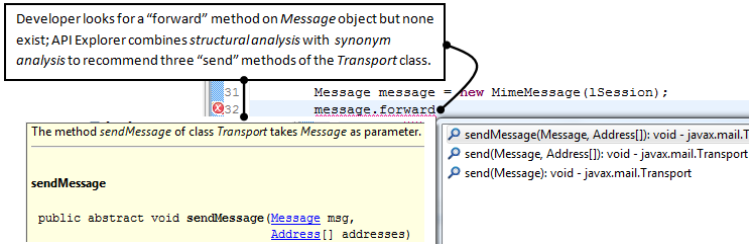


Fig. 4. API Explorer combines structural analysis with synonym analysis to recommends three send methods of the **Transport** class when a developer looks for a “forward” method on the **Message** object

was recommended, and the documentation of the recommended element to help a developer determine its relevance in the given context (see Figure 2). Once the developer makes a selection, API Explorer automatically expands the selected recommendation into code that shows how the elements needed to create an object of type **Message** should be combined (see Figure 2(B)).

API Explorer provides three options for discovering relevant methods on helper-types. With the first option, API Explorer display methods that take an object of type **Message** together with the public methods declared on **Message** through the code completion feature of the Eclipse IDE. To minimize confusion, we differentiated the recommendations of API Explorer using a different icon and appended them after the methods of the main-type. Thus, a developer browsing through the methods of **Message** using this *enhanced* code completion feature (i.e., code completion in Eclipse with API Explorer installed) will also come across the method **Transport.send(Message)**. The second option requires the developer to request explicit assistance from API Explorer. For instance, the developer would enter “message.send”, where “message” is an object of type **Message**, and API Explorer would recommend methods named “send” on other types that take an object of type **Message** as parameter. In this case, API Explorer recommended three “send” methods of the **Transport** class, and generated code of how these types should be combined once a recommendation is selected by the developer (see Figure 3). The third option handles cases where a developer might search for a method prefix that does not match the name of any method on the helper-types (e.g., searching for “message.forward” instead of “message.send”). In this case, API Explorer combines structural analysis with synonym analysis to recommend methods with a name similar to what the developer is looking for (see Figure 4). We continue in the next section with the algorithms underlying API Explorer.

4 API Graph and Recommendation Algorithms

API Explorer relies on a specialized dependency graph for APIs, called *API Exploration Graph*, and incorporates algorithms that use the information contained in the graph to generate recommendations based on the structural context.

4.1 API Exploration Graph

We use an API Exploration Graph (XGraph) to model the structural relationships between API elements. In an XGraph, API elements are represented as nodes; an edge exists between two nodes if the elements represented by the nodes share one of several structural relationships.

Nodes: an XGraph uses two kinds of nodes to represent API elements: a node to represent API types such as classes or interfaces, and a node to represent API methods. We model a public constructor as a method that returns an object of the created type.

Edges: an XGraph uses four kinds of edges to capture the relationships between API elements:

- *created-from* edge: this edge exists between an API type, T , and an API method, M , if the method M returns an object of type T . The created-from edge captures object construction through constructors, static methods, or instance methods.
- *is-parameter-of* edge: this edge exists between an API type, T , and an API method, M , if the type T is a parameter of the method M .
- *is-subtype-of* edge: this edge is used to represent subtype relationships between API types. It exists between the type T_k and the type T_m , if T_k is a subtype of T_m .
- *requires* edge: is used to distinguish instance methods from class methods. A requires edge exist between a method, M , and an API type, T , if an instance of T must exist on which the method M must be invoked.

The XGraph is simple, but by combining the information encoded in multiple edges, we are able to derive useful non-trivial facts about the relationships between API elements. For instance, from knowing that `MimeMessage` is-subtype-of `Message`, and that `Message` is-parameter-of `Transport.send`, we can infer at least three facts: first, objects of type `Message` could be created from `MimeMessage`; second, `MimeMessage` can be used whenever `Message` is expected; and third, `MimeMessage` can also be sent using the “send” method of `Transport`.

```
abstract Message {
    public void setText(String) }

MimeMessage extends Message {
    public MimeMessage(Session) }

Transport {
    public static void sendEmail(Message) }

Session {
    public static Session getInstance(Properties) }
```

Listing 1.1. A simplified version of the JavaMail API

Listing [1.1](#) shows a simplified version of the JavaMail API, and Figure [5](#) shows the corresponding XGraph. JavaMail uses the types `String` and `Properties` from the Java Runtime Environment (JRE). API Explorer maintains an XGraph of

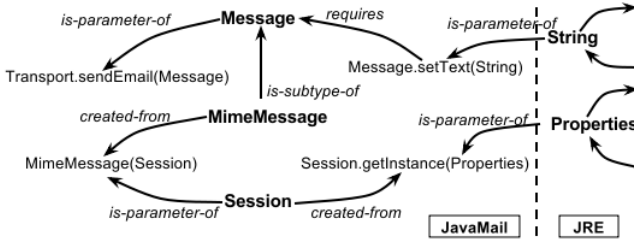


Fig. 5. The XGraph of the simplified JavaMail API in Listing 1.1. The nodes in boldface represent API types; the other nodes represent API methods, including constructors, and the edges represent relationships between the nodes.

the JRE, and automatically links it to the XGraph of APIs referencing types of the JRE, as in Figure 5. We generate XGraphs from the binaries of APIs using the Javaassist⁴ byte code analysis library, and it takes less than one minute to create an XGraph even for large APIs such as the JRE, which includes 3000 types and 9300 methods. API Explorer uses the information in the XGraph to generate recommendations and code showing how the recommended API elements should be combined. We present the recommendation algorithms in Sections 4.2 through 4.5, and use the sample API in Listing 1.1 and its XGraph to present examples of the algorithms.

4.2 Object Construction Algorithm

The object construction algorithm (Algorithm 11) facilitates the discovery of factory methods, static methods, or subtypes that may be needed to construct an object of a given API type, say T (input to the algorithm). The algorithm begins by looking at the *created-from* edges of the node representing T in the XGraph (lines 4 and 8). The `xgraph.getNodes(T , edgeType)` method (line 8) returns the API element (a factory method or constructor) each *created-from* edge points to, for every such edge found on T . The algorithm is designed to first search for a way of creating an object of type T that does not involve its subtypes. We did this to minimize the number of recommendations presented to a user. If no recommendation for creating an object of type T without its subtypes is found, the algorithm proceeds to look at the *created-from* edges of the subtypes of T (lines 9 to 12). The algorithm uses the *is-subtype-of* edge to locate the subtypes of T (line 10), then recursively calls the `getObjectConstructionProposals` method for each subtype (lines 11 to 12). The algorithm continues down the hierarchy until information on how to create an object of type of T is found, or all the subtypes are exhausted. Upon completion, the algorithm presents a list of recommendations showing different ways of creating an object of type T . We present the code generation algorithm in Section 4.5 that recursively looks for the parameters and dependencies of a selected recommendation, and generate code showing how to combine them.

⁴ www.javassist.org

Example. Consider as an example a developer looking for assistance on how to create an object of type `Message`. The algorithm begins by looking at the *created-from* edges on the `Message` node. The `Message` node has no *created-from* edge; the algorithm then proceeds by looking for subtypes of `Message` from which an object could be created. The `Message` node, in this case, has a single *is-subtype-of* edge pointing to `MimeMessage`. Next, the algorithm looks at the *created-from* edges on the `MimeMessage` node, and finds `MimeMessage(Session)`, a constructor for creating a `MimeMessage` object. The algorithm, being aware that `MimeMessage` is a subtype of `Message`, recommends `MimeMessage(Session)` as a way of creating a `Message` object.

Algorithm 1. Object Construction

```

Input: T, xgraph /* the type T for which object construction
           assistance is requested and the XGraph */
Output: recommendations /* a list of recommended API elements that
           could be used to create an object of type T */
1 Var edgeType := created-from /* a valid edge in the XGraph */
2 recommendations :=  $\emptyset$ 
3 begin
4   | recommendations := getObjectConstructionProposals(T, xgraph, edgeType)
5 end
6 function: getObjectConstructionProposals(T, xgraph, edgeType)
7 begin
8   | Var proposals :=  $\emptyset$ 
9   | proposals := xgraph.getNodes(T, edgeType) /* get the nodes in the
           XGraph pointed to by the created-from edges of node T */
10  if proposals ==  $\emptyset$  then
11    | Var subtypes = xgraph.getNodes(T, is-subtype-of)
12    | foreach type  $\in$  subTypes do
13      |   | proposals := proposals  $\cup$ 
           |   | getObjectConstructionProposals(type,xgraph,edgeType)
14  | return proposals
15 end

```

For simplicity, our example API has types with only a single object construction option. However, in practice, an API may provide multiple ways of creating objects of a given type. For instance, the JavaMail API provides four options for creating a `Session` object. In such situations, API Explorer presents all the options to a developer to decide the most appropriate construction pattern in a given usage context. As will be seen in Section 5, the participants of our case study evaluation demonstrated little difficulty selecting a relevant recommendation when presented with multiple options.

4.3 Method Recommendation Algorithm

The method recommendation algorithm (Algorithm 2) is based on the observation that if a method a developer needs is not available on the type, *T*, the

developer is working with, then one of the methods which take T , or an ancestor (a class, or an interface) of T , as a parameter may provide the needed functionality. The algorithm uses the *is-parameter-of* and the *is-subtype-of* edges of the XGraph to recommend relevant methods on other objects.

The algorithm begins by looking at the API methods that take T as a parameter using the *is-parameter-of* edges at the node T in the XGraph (lines 3, 12 to 15). The algorithm verifies if the name of a method that takes T as a parameter starts with the prefix entered by the user, and if so, adds that method to the list of proposals. If the list of proposals is empty once all the methods that take T as parameter have been examined, the algorithm uses synonym analysis to search for, and recommend, API methods with a name similar to what the developer is looking for. Our intuition is that, a developer looking for an API method to send an email object, if not searching for a method prefixed “send”, may be looking for something similar to “send”, such as “transmit” or “deliver”, instead of something totally unrelated.

Algorithm 2. Method Recommendation

```

Input:  $T$ ,  $prefix$ ,  $xgraph$  /* the type for which a recommendation is being
        requested, the prefix provided by the user, and the XGraph */
Output:  $recommendations$  /* list of recommended API methods */
1  $recommendations := \emptyset$ 
2 begin
3    $recommendations := \text{getMethodProposals}(T, prefix, xgraph)$ 
4   if  $recommendations == \emptyset$  then
5     Var  $ancestors = T.\text{getAncestors}()$ 
6     foreach  $type \in ancestors$  do
7        $recommendations := recommendations \cup$ 
8        $\text{getMethodProposals}(type, prefix, xgraph)$ 
9 end
10 function:  $\text{getMethodProposals}(T, prefix, xgraph)$ 
11 begin
12   Var  $proposals := \emptyset$ 
13   Var  $list := xgraph.\text{getNodes}(T, is-parameter-of)$  /* get the method nodes
        pointed to by the is-parameter-of edges of node T */
14   foreach  $method \in list$  do
15     if  $method.\text{nameStartsWith}(prefix)$  then
16        $proposals := proposals \cup method$ 
        /* synonym analysis */
17   if  $proposals == \emptyset$  then
18      $Set\ prefixSet = \text{getSynonyms}(prefix)$ 
19     foreach  $method \in list$  do
20        $Set\ methodSet = \text{getSynonyms}(method.\text{getName}())$ 
21       if  $methodSet \cap prefixSet \neq \emptyset$  then
22          $proposals := proposals \cup \{method\}$ 
23   return  $proposals$ 
24 end

```

The synonym analysis part of the algorithm (lines 16 to 21) re-examines all the API methods that take T as parameter. The synonym analysis begins by generating the synonym set for the prefix entered by the user (line 17); then for each method of the *is-parameter-of* edges of T , the algorithm extracts its prefix and generates its synonym set. The methods whose synonym set have one or more elements in common with the synonym set of the prefix entered by the user are added to the list of proposals (lines 20 to 21). API Explorer uses the WordNet⁵ dictionary to generate the synonym sets. We also augmented WordNet with common words such as “insert”, “put”, and “append” often used interchangeably in APIs, and by developers, but which are not necessarily synonyms in the English vocabulary.

The method recommendation algorithm may not find a relevant method amongst the methods that take T as a parameter. In this case, the algorithm searches for API methods that take an ancestor of T as a parameter (lines 4 to 8). The algorithm uses the *is-subtype-of* edges at T to locate its ancestors (line 5), and for each ancestor, calls the *getMethodProposal* method for recommendations (line 6 to 8). Upon completion, the algorithm presents a list of API methods with a prefix matching, or similar, to that entered by the developer, and with object of type T as a parameter.

Example. Consider as an example a developer looking for a “send” method on a `MimeMessage` object. The algorithm begins by looking at the *is-parameter-of* edges of the `MimeMessage` node, searching for methods prefixed “send” that take `MimeMessage` as a parameter. The `MimeMessage` node, however, has no *is-parameter-of* edge; the algorithm then looks for a supertype of `MimeMessage` by moving up its *is-subtype-of* edge, and finds the type `Message`. Next, the algorithm looks at the *is-parameter-of* edges of the `Message` node and, this time, finds an edge pointing to the static method `sendEmail(Message)` on the `Transport` class. The algorithm does not terminate once the first “send” method is found; it searches for all methods prefixed “send” that can accept a `MimeMessage` object by looking at other *is-parameter-of* edges on the current node and on other nodes up the hierarchy. In this example, the algorithm would recommend the only method it found, `Transport.sendEmail(Message)`, to the developer with the knowledge that `MimeMessage` is a subtype of `Message`.

4.4 Relationship Exploration Algorithm

In our work with APIs, we have observed cases in which a developer has identified two or more types relevant to their programming task, but remains uncertain about how these types are related [2]. Unfortunately, direct support for such an inquiry is unavailable. A developer wanting to verify the relationship between the types T_1 and T_2 must either combine the results of multiple search tools, or go through the documentation of at least one of the types before determining whether or not they are related. Using the XGraph, our relationship exploration algorithm (Algorithm 3) can help a developer efficiently explore the relationships between API types.

⁵ <http://wordnet.princeton.edu/>

The algorithm takes as input an array of API types and the XGraph, and outputs the relationships between the types, if any. Given a single API type $typeArray[0]$, the algorithm can locate other API types related to it (lines 3 to 4). The $xgraph.getRelatedTypes(typeArray[0])$ method (line 4) returns a list of types related to $typeArray[0]$ through the *is-parameter-of*, *is-subtype-of*, or the *created-from* edge of the XGraph. Given two API types $typeArray[0]$ and $typeArray[1]$, the algorithm looks for method-parameter or return type relationships between the types (lines 5 to 9). For $typeArray[0]$, the algorithm first retrieves the list of all the API methods defined on $typeArray[0]$ (line 6). Then, for each method on $typeArray[0]$, the algorithm checks whether the method takes an object of type $typeArray[1]$, or its ancestor, as a parameter, or has $typeArray[1]$, or its subtype (represented as $<:$) as a return type. If so, that method is added to the list of related elements (lines 7, 10 to 16). This same procedure is repeated for the type $typeArray[1]$ (lines 8 to 9), and the relationships between the types are presented to the user.

Algorithm 3. Relationship Exploration

```

Input:  $typeArray[]$ ,  $xgraph$  /* an array of API types and the XGraph */
Output:  $relations$  /* a list of related API element */
1 begin
2    $relations := \emptyset$ 
3   if  $typeArray.length == 1$  then
4      $relations := relations \cup xgraph.getRelatedTypes(typeArray[0])$ 
5   else if  $typeArray.length == 2$  then
6     Var  $listOfMethods0 = xgraph.getMethods(typeArray[0])$ 
7      $relations := relations \cup getRelationships(typeArray[1],listOfMethods0)$ 
8     Var  $listOfMethods1 = xgraph.getMethods(typeArray[1])$ 
9      $relations := relations \cup getRelationships(typeArray[0],listOfMethods1)$ 
10 end
11 function:  $getRelationships(T, listOfMethods)$ 
12 begin
13   Var  $relationships := \emptyset$ 
14   foreach  $method \in listOfMethods$  do
15     if  $method.getReturnType() <: T$  OR  $T \in method.getParameters()$ 
16       then
17          $relationships := relationships \cup method$ 
18   return  $relationships$ 
19 end

```

Example. Consider as an example a developer wanting to explore the relationships of `MimeMessage`. The developer will begin by issuing a query to the relationship exploration algorithm to identify the types related to `MimeMessage`. The algorithm uses the edges of the XGraph to locate types related to `MimeMessage`: in this case, the algorithm would reveal that `MimeMessage` is related to both `Message` and `Transport` using the *is-subtype-of* and the *is-parameter-of* edges of the XGraph. The developer may then explore the relationship between `MimeMessage`

and one of the related types (e.g., `Transport`) by selecting `Transport`. The algorithm then looks at the edges that connect `MimeMessage` to `Transport` in the `XGraph` to provide an explanation of how they are related. The algorithm returns within a second of the query, revealing that `MimeMessage` and `Transport` are related through the `Transport.sendEmail(Message)` method. Traditional “reference search” features, such as that provided in the Eclipse IDE, are unable to determine that `MimeMessage` is related to `Transport.sendEmail(Message)` because they are not inheritance-aware. Our relationship exploration algorithm therefore complements existing “reference search” tools.

4.5 Code Generation Algorithm

The code generation algorithm is triggered only when a recommendation is selected. This algorithm is intended to show a developer how to correctly coordinate the main-type and helper-types. If the selected recommendation is a constructor, the algorithm first determines whether or not it has parameters. If the constructor has no parameters, the algorithm generates code showing how to use the default constructor. For constructors with parameters, the code generation algorithm first generates an identifier for the non-primitive parameters, and for each non-primitive parameter `T`, calls the object construction algorithm to determine how to create an object of type `T`. The algorithm uses the method-parameter relationship to determine how the statements should be ordered and how they relate to each other.

If the selected recommendation is an API method, the algorithm uses the *requires* edge to determine whether or not the method is static. For a non-static method, the algorithm begins by calling the object construction algorithm to create an object of the type on which the method is defined, before invoking it. Then, for each non-primitive parameter `T` of the selected method, the code generation algorithm calls the object construction algorithm to determine how to create an object of type `T`. For a static API method (i.e., method without a *requires* edge), the algorithm only has to create objects for each non-primitive parameter. The algorithm does not create objects for non-primitive parameter types already available from the context in which API Explorer was invoked — it uses variables in the context that match a given parameter type. For instance, if a developer selects `Transport.send(Message)` from the recommendations on how to send a `Message` object `m1`, the code generation algorithm will not create a new `Message` object, but will pick `m1` from the context, and output `Transport.send(m1)`.

4.6 Design Rationale

We designed our approach with the awareness that a main-type may have several helper-types, with each helper-type relevant to a different programming scenario. For instance, the type `Message` of the JavaMail API has the method `Transport.send(Message)` as a helper-type for sending email objects, and the method `SearchTerm.match(Message)` as a helper-type for locating email objects that satisfy a given search criterion. Similarly, an API type may have several

object construction patterns, with each pattern relevant to a different usage scenario. Our approach does not attempt to guess which helper-type is relevant for a given programming scenario; it recommends all valid helper-types in a given structural context, and allows the developer to select the most appropriate helper-type for a given programming scenario. We designed our approach this way for two reasons: first, a heuristic that attempts to narrow down the list of recommended help-types by removing those considered irrelevant in a given scenario may inadvertently hide a helper-type most appropriate for a given scenario. Such a mistake will further undermine discoverability, the very problem our approach is intended to solve. To avoid hampering discoverability, we opted for a design that relies on the developer to select the helper-type most appropriate for a given task. Second, our experience working with APIs indicates that developers have little problem selecting relevant API elements from a list of recommendations. We therefore expect that developers will have little difficulty selecting the most appropriate helper-type from a list of possible helper-types for a given programming task. We discuss the extent to which our expectations were valid in Section 5.

5 Evaluation

Our evaluation had two goals: first, to show the extent to which our assumptions about the API exploration behavior of developers, and their ability to select relevant recommendations, are reflected in realistic API usage contexts; and second, to understand the contexts in which API Explorer may be helpful in discovering helper-types not accessible from a given main-type. Given that we were interested in studying how the approach supports people (as opposed to the performance of algorithms taken in isolation), we favored a qualitative evaluation methodology. We reasoned that a qualitative evaluation of our approach in the context of several programming tasks will enable us to reliably evaluate the assumptions and observations on which our approach is based, and to understand the contexts in which the approach would not be effective.

5.1 Case Study Design

We used a case study methodology to evaluate our approach. Yin introduces the case study methodology as “an empirical inquiry that investigates a contemporary phenomenon within its real-life context” [20, p. 13], and Easterbrook et al. explains that the case study methodology is particularly suited for evaluating software tools “where the context is expected to play a role in the phenomena” [3, p. 297], as in the case of API Explorer. For example, Holmes and Murphy used a case study evaluation to provide an in-depth understanding of how and why their Strathcona tool was helpful, ICSE ’05 [6].

In the case study methodology, the cases (programming tasks, in our setting) are selected to represent the phenomenon being studied, and each case is considered as a replication, rather than a member of a sample [3,20]. Furthermore, our case study methodology emphasizes generalization to similar contexts (i.e.,

if the selected cases supports our hypotheses, then it is expected that similar cases will be supported by our approach), not statistical generalization [20, p. 31]. The goal of our case study was to answer the following questions:

Q.1 To what degree are our assumptions about the API exploration behavior of developers reflected in practice?

Q.2 In which ways can structural relationships help when trying to increase the discoverability of API elements necessary to solve a task?

Q.3 Would a developer be able to select a helper-type relevant to their task when presented with a list of possible helper-types?

Q.4 In which situations would API Explorer not be helpful, and why?

A. Programming tasks

Our approach is intended to assist developers locate helper-types not accessible from a type they may be working with. We therefore selected programming tasks that typified the discoverability hurdles our approach is intended to solve. Three of the tasks (the Email, XML, and Chart tasks) selected for the study have been the subject of previous studies that investigated the discoverability problem [2][16].

Email task: we asked the participants to use the JavaMail API to implement a solution that would compose and deliver an email message. To complete the task, a participant needed to create and configure at least four API types, all created from factory methods or subtypes, and needed to discover a key relationship between `Message` and `Transport` to send the email message. We used version 1.4.2 of the JavaMail API, which has five packages and 91 non-exception classes.

XML task: we asked the participants to use the Java API for XML Processing (JAXP)⁶ to verify whether the structure of an XML file conforms to a given XML schema file. This task required the combination of at least four API types (`Validator`, `Schema`, `SchemaFactory`, and `Source`); we selected this task to evaluate the object construction feature because of the unique challenges it presents — all the required types are abstract with no subtypes; the types must be created from factory or public methods (e.g., `Validator` can only be created from `Schema.newValidator()`). We used version 1.4 of the JAXP API, which has 23 packages and 207 non-exception classes.

Chart task: we asked the participants to use the JFreeChart⁷ API to create a pie chart and to save the chart to a file in a graphic format. To complete this task, a participant needed to coordinate at least five API types, and had to discover the relationship between `JFreeChart`, the type for representing charts, and `ChartUtilities`, the type needed to save the chart. We used version 1.0.13 of the JFreeChart API, which has 37 packages and 426 non-exception classes.

PDF Task: we asked the participants to use the PDFBox⁸ API to implement a solution to merge two PDF files. This task required the combination of just two

⁶ jaxp.dev.java.net

⁷ jfree.org/jfreechart

⁸ pdfbox.apache.org

API types: `PDFDocument` and `MergerUtility`. However, the relationship between `PDFDocument` and `MergerUtility` (related through an “append” method on `MergerUtility`) cannot be determined through synonym analysis since “merge” is not a synonym of “append”. We were interested in investigating whether the participants would be able to use other features of API Explorer to discover this key relationship. We used version 1.2.1 of the PDFBox API, which has 31 packages and 307 non-exception classes.

B. Study participants.

We recruited eight participants (henceforth referred to as P1, ..., and P8) through our departmental mailing list. Our participants reported between 1.5 and 3 years of experience programming with Java, with a median Java programming experience of 2.5 years. All the participants had at least six months experience working with the Eclipse IDE. None of the participants, with the exception of P1, had used any of the four APIs in the study; P1 had used the `JFreeChart` API in the past, but in a task different from ours and could not remember the types provided by the API.

C. Study procedure.

We provided each participant with a tutorial of the features of API Explorer before the study began, and asked the participants to use API Explorer whenever they believed a feature it provides could be helpful. We also provided each participant with a description of the tasks and the documentation of the APIs. The four tasks were completed in the same order by the participants, and the participants were allowed a maximum of forty minutes per task. We asked the participants to think-aloud whenever API Explorer was used to allow us to understand why the assistance of API Explorer was needed, why the participant selected a given recommendation, and whether or not the assistance provided by API Explorer was helpful. We also used screen capturing software to document all the actions of the participants. To avoid influencing the behavior of the participants, we did not inform them of which types of an API were relevant to each task, or which type of an API to start from; the decision of how to approach each task was left to each participant.

5.2 Results

The study produced a total of over 16 hours of screen captured videos and verbalizations of eight participants using API Explorer in 32 different programming sessions. Our analysis of the data from the study focused on the questions the study was designed to answer. We begin by presenting task-level observations that show the degree to which the API exploration behavior of the participants supports the hypothesis on which our approach is based (Q.1). For each task, and for each participant, we provide observations on how the participant approached the task, and the degree to which API Explorer was effective in helping the participant discover helper-types not accessible from a main-type. Then, we present episode-level observations: an analysis of all the instances in which API Explorer was used by each participant, the degree to which a participant was able to select relevant recommendations, and the discoverability contexts in which API

Table 1. A summary of the results of how the participants approached each task, their effectiveness in using API Explorer (APIX) to locate helper-types not accessible from a main-type, and the API Explorer feature (SA — synonym analysis, EC — enhanced code completion, RE — relationship exploration, OC — object construction) used to make the discovery. The check mark (✓) represents Yes, and ✗ represents No.

	P1	P2	P3	P4	P5	P6	P7	P8
Email Task								
Started from <i>Message</i> , then looked for <i>Transport</i>	✓	✓	✓	✓	✓	✓	✓	✓
Found <i>Transport.send</i> from <i>Message</i> using APIX	✓	✓	✓	✓	✓	✓	✓	✓
Feature used	EC	SA	SA	SA	EC	SA	SA	EC
Chart Task								
Started from <i>JFreeChart</i> , then looked for <i>ChartUtil</i>	✓	✓	✓	✓	✓	✗	✓	✓
Found <i>ChartUtil.write</i> from <i>JFreeChart</i> using APIX	✓	✓	✓	✓	✗	✗	✓	✓
Feature used	EC	SA	SA	EC	—	—	EC	SA
PDF Task								
Started from <i>PDFDoc</i> , then looked for <i>MergerUtil</i>	✓	✓	✓	✓	✓	✓	✓	✓
Found <i>MergerUtil.append</i> from <i>PDFDoc</i> using APIX	✓	✓	✓	✓	✓	✓	✓	✗
Feature used	EC	RE	EC	EC	EC	EC	EC	—
XML Task								
Started from <i>Validator</i> , then looked at <i>Schema</i>	✗	✓	✓	✓	✗	✓	✓	✗
Found <i>Schema.newValidator()</i> from <i>Validator</i> using APIX	✓	✗	✓	✓	✓	✗	✓	✓
Feature used	OC	—	OC	OC	OC	—	OC	OC

Explorer proved helpful (Q.2 and Q.3). Lastly, we look at situations in which API Explorer was not helpful (Q.4).

A. Tasks-Level Observations.

The first question (Q.1) was intended to investigate the degree to which the behavior of our participants supports our main hypothesis (*when working on a task, a developer typically starts from a main-type central to the programming scenario before looking for helper-types*) and to evaluate the degree to which API Explorer would be helpful in discovering relevant helper-types. Due to space restrictions, we present a detailed outline of the observations from the Email task, and summarize the observations from the other tasks in Table 11.

⁹ A detailed outline of how the participants approached each task, and how they used API Explorer is available at: www.cs.mcgill.ca/~swevo/explorer/evaluation/

All eight participants started the Email tasks by looking for a type representing an email message. They all found the abstract class `Message` from the documentation and proceeded to query API Explorer for assistance on how to create an object of type `Message`. API Explorer provided two recommendations: `MimeMessage(Session)` and `MimeMessage(Session,InputStream)`, both constructors from the subtype `MimeMessage`; seven of the participants selected `MimeMessage(Session)`, P5 selected `MimeMessage(Session,InputStream)` thinking `InputStream` is needed to set the email content. P5 later reverted to `MimeMessage(Session)`. After selecting `MimeMessage(Session)`, API Explorer provided four recommendations on how to create a `Session` object from factory methods, and all the eight participants selected `Session.getInstance(Properties)`, to complete the process of creating a `Message` object.

The participants approached the next part of the task, sending the email message, differently. P1 started with the documentation in search for assistance on how to send the message but did not find `Transport`. He then browsed through the methods of `Message` using the *enhanced* code completion (EC) feature of Eclipse when he noticed `Transport.send(Message)` amongst the recommendations of API Explorer. P5 and P8 also used the EC to discover `Transport.send(Message)` directly from `Message`. Participants P2, P3, P4, P6, and P7 all used the synonym analysis (SA) feature of API Explorer to query for a recommendation for “`Message.send`”, and received four recommendations from which they discovered three different “send” methods on the `Transport` class.

We present a summary of the observations from the other tasks in Table III. For each task, we indicate whether the participant started from the main-type before looking for the helper-type, whether the participant was able to use API Explorer (APIX) to discover the helper-type directly from the main-type, and the API Explorer feature that was used to make the discovery. For the Chart task, seven of the eight participants started from the main-type `JFreeChart` before looking for the helper-type `ChartUtilities`. Only P6 started from `ChartUtilities` before looking for `JFreeChart`, and this occurred because P6 had difficulties finding the main-type and happened to stumble on `ChartUtilities`. Six of the eight participants successfully used APIX to discover `ChartUtilities` directly from `JFreeChart`. P5 did not attempt to use APIX to look for a helper-type; he came up with an improvised solution that created a `BufferedImage` from `JFreeChart`. For the PDF task, all the eight participants started from the main-type `PDFDocument` before looking for the helper-type `MergerUtility`, and seven of the participants successfully used APIX to discover `MergerUtility` directly from `PDFDocument`. P8 used synonym analysis with “`PDFDocument.merge`” but got no recommendations. He made no attempt to use other features of APIX, such as the enhanced code completion, that could have helped him discover `MergerUtility`; he came up with an improvised solution for merging the documents.

Five of the eight participants in the XML task started with the main-type `Validator`; the other three started with the helper-type `Schema`. The domain that provided support for validation had only six classes, with `Schema` at the top of the list, and `Validator` at the end: that could have influenced the three participants

that started with `Schema`. Six of the participants used the object construction feature to discover how to create a `Validator` object from `Schema.newValidator()`, the other two used the documentation.

The results were consistent across the eight participants and in most of the tasks: the participants typically began exploring an API from the main-type before looking for a relevant helper-type, and successfully used API Explorer to discover relevant helper-types directly from a main-type.

B. Episode-Level Observations.

To answer questions Q.2, Q.3, and Q.4, we analyzed all the segments of the screen captured videos, which we called *episodes*, corresponding to instances in which a participant used API Explorer to discover API elements relevant to a task. In our analysis, we focused on the degree to which a participant was able to select API elements relevant to a task from the recommendations of API Explorer, the discoverability contexts in which the assistance of API Explorer was requested, and whether or not the assistance provided was helpful. We consider the assistance provided by API Explorer *helpful* if its recommendations contains an API element relevant to a given request, and if the participant was able to recognize and select the relevant element. The results of the analysis are summarized in Table 2.

The third column (# of usage episodes) of Table 2 shows the number of episodes where API Explorer was used, per participant and per discoverability context. For instance, P1 used API Explorer 21 times: four times to discover relevant methods on other API types (row METH), 16 times to discover API elements necessary to construct an object of a given API type (row OBJ), and once to look for types related to a given API type that could be used to perform a given operation (e.g., types related to `PDFDocument` that could be used for merging; row ER). The participants requested the assistance of API Explorer a combined total of 161 times. The fourth column presents the average number of recommendations per episode for each of the different discoverability contexts. The average number of recommendations ranged from about 2 to 15 recommendations per episode.

The fifth column presents the number of episodes in which a participant was *unable* to select or recognize an API element relevant to a task from the recommendations made by API Explorer. We observed only two instances in which a participant was unable to select a relevant API element from the recommendations of API Explorer. In the first instance, P3 had requested the list of API types related to `PDFDocument` while looking for a type that could be used for merging PDF files. API Explorer provided a list with 12 API types, including `MergerUtility`, but P3 failed to notice it because it was not visible, and P3 did not scroll to examine the entire list. In the second instance, P7 had requested for assistance on how to create a `Schema` object, and received eight recommendations: P7 selected `DocumentBuilder.getSchema()` instead of `SchemaFactory.newSchema(File)`, but later reverted to `SchemaFactory.newSchema(File)` when she realized a schema file was provided for the task. API Explorer

Table 2. A summary of all the instances in which API Explorer was used by each participant for the various contexts (object construction [OBJ], looking for relevant methods on other types [METH], and exploring the relationships between types [ER])

		# of usage episodes	average # of recommendations	unable to select	API Explorer not helpful
P1	OBJ	16	6.3	0	0
	METH	4	5	0	0
	ER	1	0	0	1
P2	OBJ	12	5.5	0	0
	METH	4	4.2	0	2
	ER	4	6	0	1
P3	OBJ	11	7.8	0	0
	METH	3	8.2	0	0
	ER	8	3.5	1	1
P4	OBJ	16	6.3	0	0
	METH	3	9.1	0	0
	ER	5	5.9	0	0
P5	OBJ	14	7	0	0
	METH	2	15.2	0	0
	ER	3	0	0	0
P6	OBJ	12	8.3	0	0
	METH	4	5.1	0	1
	ER	5	3.6	0	0
P7	OBJ	11	7.1	1	1
	METH	3	8.6	0	0
	ER	1	5.5	0	0
P8	OBJ	14	6.2	0	0
	METH	2	8.4	0	0
	ER	3	1.7	0	0
TOTAL		161		2	7

was not helpful in only seven of the 161 episodes in which it was used (last column): we address these situations below where we look at the limitations of our approach.

The participants experienced little difficulty selecting API elements relevant to a given programming scenario when presented with a list of possible helper-types. API Explorer also proved mostly helpful when looking for helper-types relevant to creating an object, relevant helper-methods on other objects, and when looking for types related to a given API type that could be used to perform a given operation.

C. Limitations of our approach

Our approach will not be helpful if the relationships between API elements can only be determined at runtime. The last column of Table 2 shows other situations in which our approach was not helpful: these involve the synonym analysis and relationship exploration features of API Explorer.

The effectiveness of our synonym analysis algorithm depends on API methods respecting naming conventions such as method names beginning with action verbs, not acronyms, and on the ability of a developer to provide a prefix that match, or is a synonym to the name of a relevant method on a helper-type. In two instances, P2 and P8 had sought for assistance on how to merge PDF files using synonym analysis with “PDFDocument.merge” but received no recommendation. This was expected as “merge” is not a synonym of the “append” method on `MergerUtility`. We had designed the PDF task to see whether the participants would be able to use other features of API Explorer to discover `MergerUtility` from `PDFDocument`. In particular, to address the limitations of the synonym analysis feature, we *enhanced* the default Eclipse code completion feature with the ability to display not only the methods defined on type T , but also the API methods that take an object of type T as a parameter. For instance, a developer browsing through the methods of `Message` using this enhanced code completion feature will also come across the method `Transport.send(Message)`. Thus, a relevant helper-method not recommended by synonym analysis will be discovered when the developer looks through the methods of T . As shown in Table 1 (PDF task), six of the eight participants were able to use the enhanced code completion feature to discover `MergerUtility` directly from `PDFDocument`.

Our relationship exploration algorithm has two limitations: it can not identify the API types that throw a given exception, and can only identify direct relationships between API types. P1 had looked for types related to `SendFailedException` that could be used to send an email message but was misinformed that there was no related type, although this exception is thrown by `Transport`. This occurred because the current version of our XGraph does not support types related through thrown exceptions. However, P1 subsequently discovered `Transport.send` with the assistance of the method recommendation feature of API Explorer. P2 was misinformed that `Document` is not related to `Source`, although they are related through `DOMSource(Document)`, a constructor of a subtype of `Source`. This occurred because our relationship exploration algorithm does not consider indirect relationships between API elements. We plan on extending our XGraph and algorithms to show API types that throw a given exception and to support indirect relationships between API elements.

5.3 Summary

Overall, the results of the study were consistent across the participants and for most of the tasks: the participants began exploring the APIs from a main-type before looking for the helper-types, and were mostly successful at using API Explorer to locate helper-types not accessible from a main-type. The participants also experienced little trouble selecting relevant elements when presented with multiple recommendations. Our understanding of the domain enabled us

to select tasks from real-world APIs with discoverability hurdles typical to those that have been identified in the literature [416]. We therefore expect our observations to generalize to similar contexts, namely, when seeking to make API elements not directly accessible from a given API type more discoverable. The participants expressed four reasons why they considered the assistance provided by API Explorer helpful:

- *Saves time* (P2, P3, P4, P5, P7, P8): “It would have taken me a lot of time to go [to the documentation] and find which class will have a merge functionality. Using the tool, I could find `MergeUtility` directly from `PDFDocument`” – P2.
- *Increases awareness* (P1, P4, P6, P7, P8): “this is another thing I really like. A lot of times when you look at an API, you look at just the first constructor and use that. API Explorer shows me other better options that I wouldn’t have looked for.” – P1.
- *Serves as a reminder* (P1): “I couldn’t remember the proper way of using it [the `JFreeChart` class] and was reminded by the tool” – P1.
- *Unmasks hidden relationships* (P1, P2, P4, P5, P7, P8): “If you want to save something, you would like to say `object.save()` but that option is usually not provided; usually, it is `something.save(object)` [that is provided]. It [API Explorer] is useful because it can make the association between the object you want to save and the method that you need to call” – P1.

API Explorer recursively shows a participant how to create and relate objects necessary to use a selected recommendation, even if the required objects comes from commonly used types. Two participants (P4 and P6) complained that this was not necessary for commonly used types such as the `String` class: “telling me how to construct a `String` might not necessarily be the most helpful thing because it is commonly used.” – P6.

5.4 Threats to validity

As indicated in Section 5.1, our method of choice for evaluating API Explorer was the case study, which emphasizes exploration of the relation between a phenomenon and its context as opposed to generalization. In particular, the diversity of APIs and programming languages present factors which limits the generalizability of the results of our study. First, API Explorer will not be helpful for APIs without helper types, or APIs without indirect object construction patterns such as the Factory pattern. The same is true for an API with a well-written API documentation that include actual usage examples. History, however, suggests that we are far from these ideals: there are situations where it seems reasonable to provide a Factory, instead of a constructor, and to provide helper-types. It is for such situations that we envisage tools such as API Explorer to remain helpful in facilitating discoverability in APIs. Second, although the tasks used in our evaluation were drawn from real-world APIs, it is likely that they did not uncover every discoverability hurdle that could occur in practice. In particular, very few indirect relationships, a feature not currently supported by API Explorer, were uncovered by the evaluation. As future work, we plan on extending API Explorer to support indirect relationships and to conduct further studies

to evaluate this feature. Lastly, some APIs have the notion of an internal API, intended to be used by the designers only, and the public API, for general use. The current version of API Explorer does not take these differences in account when making recommendations; there is therefore the possibility that recommendations made by API Explorer may be from the internal API, a practice discouraged by API designers.

6 Related work

Improving Code Completion Tools. Previous work on code completion systems focused either on re-ordering the list of methods accessible on a given type, or on predicting the method of an API type most likely to be called next in a given context. Robbes et al. modeled the change history of systems as atomic operations and used this history to predict the method of an object most likely to be called next in a given context [11]. Bruch et al. used example code in code repositories to improve the ordering of the list of suggested methods [1]. These previous works can only suggest or re-order elements accessible on the object on which code completion is requested. API Explorer is a novel extension of code completion, capable of suggesting relevant methods on other objects, and providing support for locating elements relevant to the use of a given API type.

Other IDE Tools. IDEs provide tools that could be used to search for places where an API type is referenced, and potentially, locate elements not structurally accessible from a given type. These tools are suited for code comprehension, not API exploration, and there is no evidence that a participant from any of the previous studies even attempted to use these tools when learning how to use APIs [24][15][16]. On the contrary, observations from previous studies indicated that the content assist feature is the most widely used when exploring APIs [15][16].

Documentation Improvement Tools. Some efforts on facilitating the discovery of API elements have focused on improving the API documentation. Kim et al. proposed eXoaDocs [9], a tool that integrates code snippets mined from source code search engines into the Java API documentation, making factory methods or subtypes necessary for object construction discoverable. Jadeite [17] uses usage statistics of the types and methods of an API from code examples found on the Web to help developers find commonly used API elements from the documentation, and also integrates code snippets on how to construct objects of API types in the documentation. Jadeite also has a concept, similar to our method recommendation feature, known as a “placeholder” which allows a developer to annotate the documentation with the name of a method expected to be located on a given API type, and to link the “placeholder” to an actual method of the API that should be used instead. API Explorer, in contrast, automatically identifies relevant methods on other API types using the structural relationships between API elements, and presents this information through the code completion feature of the IDE. Furthermore, Jadeite and eXoaDocs require large collections of example usages of APIs. API Explorer, in contrast, is

lightweight, leveraging the structural relationships between API elements, not collections of code examples, to make API elements discoverable.

Example Recommender Tools. These tools leverage the proliferation of code examples on the Web and open-source repositories to make learning how to use APIs easier; they differ in the approach used to retrieve code examples and in the kind of support afforded to API users. CodeBroker [19] uses comments and method signatures written by the programmer to recommend methods from code repositories. Strathcona [6] uses the structural context of the code under development such as the parent class of the framework type being extended, and the signature of API methods to retrieve relevant code examples from a repository. MAPO uses pattern mining techniques to identify code snippets and method call sequence that show how to use a given API method [21]. Prospector [10], ParseWeb [18], and XSnippet [13] take queries of the form “`source-type` → `destination-type`”, and recommend code examples that show how to get the `destination-type` from the `source-type`. Jiang et al. [8], Salah [14], and Heydarnoori [5] proposed tools which use dynamic analysis of the interaction between sample applications and APIs to identify valid usage scenarios and valid call sequence of API methods. Code Conjuror [7] uses test cases written by programmers to retrieve example usages of APIs element from code repositories.

Prospector and XSnippet are the most similar to API Explorer because they combine the use of code examples with the structural relationships between API elements such as return types and method parameters to identify relevant method call sequences that link a `source-type` to a `destination-type`. However, the support provided by Prospector and XSnippet is limited to object construction only, and for both tools to work, a developer is expected to provide both a `source-type` and a `destination-type`. As observed in our case study, and also in a previous API usability study [2], a developer may not even be aware of the necessary `destination-type`. With API Explorer, developers can obtain object construction support with only a `source-type`. Furthermore, API Explorer extends these works by using structural relationships to make relevant API methods not accessible from an API type discoverable.

7 Conclusion

Learning how to use APIs is major part of a software developer’s job. Even experienced developers must learn newer parts of an existing API, or newer APIs, when working on a new project. This paper addresses one of the challenges developers face when learning a new API: discovering relevant helper-types not accessible from a main-type they are working with. We have proposed an approach that leverages structural relationships to make relevant API elements not accessible on a given API type discoverable. We implemented our approach in a tool called API Explorer, and evaluated the approach through a multiple-case study in which eight participants replicated four programming tasks with several discoverability hurdles. The results of the study was consistent across the participants and the tasks: API Explorer effectively assisted the participants

to locate helper-types not accessible from a main-type in different discoverability contexts. The participants also experienced little difficulty selecting relevant API elements from the recommendations of API Explorer. The results provide initial evidence that the use of structural relationships to make API elements discoverable could be a viable, and an inexpensive, alternative to API wrappers or API restructuring when seeking to improve discoverability in APIs.

References

1. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: *Proceedings of the 7th Joint ESEC/FSE*, pp. 213–222 (2009)
2. Duala-Ekoko, E., Robillard, M.P.: The information gathering strategies of API learners. Technical report, TR-2010.6, School of Computer Science, McGill University (2010)
3. Easterbrook, S., Singer, J., Storey, M.-A., Damian, D.: Selecting empirical methods for software engineering research. In: *Guide to Advanced Empirical Software Engineering*, pp. 285–311. Springer, Heidelberg (2008)
4. Ellis, B., Stylos, J., Myers, B.: The factory pattern in API design: A usability evaluation. In: *Proc. of the 29th International Conf. on Software Eng.*, pp. 302–312 (2007)
5. Heydarnoori, A.: Supporting Framework Use via Automatically Extracted Concept-Implementation Templates. PhD thesis, School of Computer Science, University of Waterloo (2009)
6. Holmes, R., Murphy, G.C.: Using structural context to recommend source code examples. In: *Proc. of the 27th International conf. on Software Eng.*, pp. 117–125 (2005)
7. Hummel, O., Janjic, W., Atkinson, C.: Code conjurer: Pulling reusable software out of thin air. *IEEE Software* 25, 45–52 (2008)
8. Jiang, J., Koskinen, J., Ruokonen, A., Systa, T.: Constructing usage scenarios for API redocumentation. In: *Proc. of the 15th International Conf. on Program Comprehension*, pp. 259–264 (2007)
9. Kim, J., Lee, S., Hwang, S.W., Kim, S.: Adding examples into java documents. In: *Proc. of the International Conf. on Automated Software Eng.*, pp. 540–544 (2009)
10. Mandelin, D., Xu, L., Bodík, R., Kimelman, D.: Jungloid mining: helping to navigate the API jungle. In: *Proc. of the International Conf. on Programming Language Design and Implementation*, pp. 48–61 (2005)
11. Robbes, R., Lanza, M.: How program history can improve code completion. In: *Proc. of the 23rd Conference on Automated Software Eng.*, pp. 317–326 (2008)
12. Robillard, M.P., DeLine, R.: A field study of API learning obstacles. *Empirical Software Engineering* (to appear, 2011)
13. Sahavechaphan, N., Claypool, K.: Xsnippet: mining for sample code. In: *Proceedings of the 21st OOPSLA*, pp. 413–430 (2006)
14. Salah, M., Denton, T., Mancoridis, S., Shokoufandeh, A., Vokolos, F.I.: Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In: *Proc. of the 21st International Conf. on Software Maintenance*, pp. 155–164 (2005)
15. Stylos, J., Clarke, S.: Usability implications of requiring parameters in objects’ constructors. In: *Proc. of the 29th International Conf. on Software Eng.*, pp. 529–539 (2007)

16. Stylos, J., Myers, B.A.: The implications of method placement on API learnability. In: Proc. of the 16th International Symposium on Foundations of Software Eng., pp. 105–112 (2008)
17. Stylos, J., Myers, B.A., Yang, Z.: Jadeite: improving API documentation using usage information. In: Extended Abstracts on Human Factors in Computing Systems, pp. 4429–4434 (2009)
18. Thummalapenta, S., Xie, T.: Parseweb: a programmer assistant for reusing open source code on the web. In: Proc. of the 22nd International Conf. on Automated software Eng., pp. 204–213 (2007)
19. Ye, Y., Fischer, G., Reeves, B.: Integrating active information delivery and reuse repository systems. In: Proc. of the 8th International Symposium on Foundations of Software Eng., pp. 60–68 (2000)
20. Yin, R.K.: Case Study Research: Design and Methods, 2nd edn. Sage, Thousand Oaks (2003)
21. Zhong, H., Xie, T., Zhang, L., Pei, J., Mei, H.: MAPO: Mining and recommending API usage patterns. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 318–343. Springer, Heidelberg (2009)

Mining Evolution of Object Usage

Yana Momchilova Mileva, Andrzej Wasylkowski, and Andreas Zeller

Saarland University, Saarbrücken, Germany
{mileva,wasykowski,zeller}@cs.uni-saarland.de

Abstract. As software evolves, so does the interaction between its components. But how can we check if components are updated consistently? By abstracting object usage into temporal properties, we can learn *evolution patterns* that express how object usage evolves over time. Software can then be checked against these patterns, revealing code that is in need of update: “Your check for `isValidWidget()` is now superseded by `checkWidget()`.” In an evaluation of seven different versions of three open source projects, our LAMARCK tool was able to detect existing code issues with a precision of 33%–64% and to prevent such issues with a precision of 90%–100%.

1 Introduction

In software development, change is the only constant. New features are added, defects are fixed, or code is refactored to improve maintainability. In each of those cases, changes must be *consistently applied* to avoid defects and maintenance problems.

As an example of a project-wide change, consider the example Eclipse method `removeSelectionListener()` shown in Figure 1. In Eclipse 1.0, this method calls `isValidThread()` and `isValidWidget()` to verify that its preconditions are satisfied. In Eclipse 2.0, however, these two calls have been replaced with a call to `checkWidget()`—a new method which encompasses the two original checks and which can easily be extended to implement additional checks. This change has been applied across several Eclipse 2.0 methods that performed similar checks. But how do we know we found them all? And how do we ensure that new code actually uses `checkWidget()` rather than falling back to the old style?

In this paper, we address these problems and introduce a tool that solves them, called LAMARCK¹. LAMARCK analyzes the *changes* that occurred between two versions of the same project, determines the *object usage* in both versions and derives *evolution patterns*—that is, changes that have been consistently applied at multiple locations, like the one in Figure 1. These patterns can be forwarded to developers, informing them of object usage changes. As formal descriptions,

¹ Jean-Baptiste Lamarck (1744–1829) was an early proponent of organic evolution, proposing that organisms became transformed by their efforts to respond to the demands of their environment. He was, however, unable to explain a mechanism for this. [13, under “Lamarck” and “evolution”].

```

void removeSelectionListener(Listener listener){
-   if (!isValidThread ())
-       error (SWT.ERROR_THREAD_INVALID_ACCESS);
-   if (!isValidWidget ())
-       error (SWT.ERROR_WIDGET_DISPOSED);
+   checkWidget();
    if (listener == null)
        error (SWT.ERROR_NULL_ARGUMENT);
    if (eventTable == null) return;
    eventTable.unhook (SWT.Selection, listener);
    eventTable.unhook (SWT.DefaultSelection,listener);
}

```

Fig. 1. Method change from Eclipse 1.0 to 2.0. The old check (prefixed with “-”) has been replaced by a custom method call (prefixed with “+”). Has this change been propagated consistently across Eclipse?

however, they can also be used to detect *violations*—places in the code where a change should have been applied, but was not.

We have evaluated LAMARCK on seven different versions of three open source projects. LAMARCK found over 2000 evolution patterns followed by those projects. As it comes to *preventing errors*, LAMARCK was able to detect missing changes with a precision of 90%–100%. Every time, LAMARCK provided a crisp and accurate representation of the change to be applied. On top of that, LAMARCK can also be helpful for *detecting* errors in existing code, uncovering complex API usage changes with a true positive rate of 33%–62%. In the preventing errors scenario, LAMARCK can be used to assist developers apply the needed changes and warn them against violating a pattern, while in the detecting errors scenario the developers have in a sense decided that they have corrected all the locations that need change—then LAMARCK looks automatically for such omitted locations.

The contributions of our work are as follows:

1. The first approach to study how object usage changes over time;
2. The first approach to combine specification mining with mining source code archives;
3. A novel approach to detect missing and incomplete changes, based on object usage;
4. A novel approach to detect bugs due to inconsistent API usage, based on API evolution.

The remainder of this paper follows the flow of information through our LAMARCK tool (Figure 2). In Section 2, we introduce the concept of *software evolution* from the perspective of evolving object usage. Section 3 introduces the concept of *evolution patterns* and shows how they and their violations (i.e., missing changes) can be detected. In Section 4, we present our evaluation on *detecting* and *preventing* errors. We conclude the paper with the survey of related work (Section 5), and close with conclusions and future work (Section 6).



Fig. 2. How LAMARCK works. Given two versions (a), LAMARCK extracts temporal properties (b) that characterize object usage in each version. The differences (c) are then mined for recurrent change patterns (d). These patterns can then be used to search for missing changes (e) in new code (f).

2 Object Usage Evolution

As stated initially, **our goal is to have changes applied consistently across a piece of software**. To apply changes consistently, we need a notion of *similarity*—a similarity of changes, but also of contexts in which these changes are to be applied. Choosing an appropriate abstraction level for similarity is tricky. If we choose it too low, we end up with *syntactic* similarity, where changes are deployed consistently only across copy-and-paste clones. If we choose it too high, we end up with *semantic* similarity, which is generally undecidable.

To characterize changes, we use an abstraction that is well understood in software design—application programming interfaces (APIs). The key idea is to monitor how *the usage of object APIs evolves in individual versions*. This allows us to abstract away from syntactic similarity, yet detect inconsistent evolution in object usage: If a component still interacts with an API in a deprecated fashion, it is in need of an update.

Of course, there are established concepts to deter usage of “old” APIs: Individual functions may be marked as “deprecated”, causing a warning during compilation time, or simply not offered at all. However, “old” API usage may not necessarily mean using “old” functions, it may be a specific “old” combination of existing functions that is no longer up to date. In Figure 1, the methods `isValidThread()` and `isValidWidget()` still exist in Eclipse 2.0, and are still being used; it is this specific usage in this specific context, though, that now has evolved. Any characterization of object usage thus must express *relationships* between individual functions—to characterize both the evolving APIs as well as the context into which they are embedded.

In earlier work [17], Wasylkowski, Zeller, and Lindig described a formalism that satisfies these requirements—so-called *temporal properties* of object usage. To express the fact that method `a()` may be used before method `b()`, they use the syntax `a() < b()`. The temporal properties for the Eclipse 1.0 version of the code shown in Figure 1, for instance, include `isValidThread() < isValidWidget()` and `isValidThread() < error()`.

However, temporal properties can also carry *dataflow* information, denoting objects that are *shared* across these properties. The term “return value of `a()` <

second argument of `b()`” means that the return value of `a()` is used as the second argument of `b()`. In the Eclipse 1.0 version in Figure 11, the properties thus actually read “target of `isValidThread()` \prec target of `isValidWidget()`” and likewise, because the two methods share the same target object (the implicit `this` object).

If the API usage changes, the temporal properties will also change. Since temporal properties encode dataflow, such changes will also characterize changes in argument ordering, or changes in the order of method calls. In Figure 11, for instance, the properties “target of `isValidWidget()` \prec target of `error()`”, “target of `isValidThread()` \prec target of `error()`”, and likewise, will be replaced by “target of `checkWidget()` \prec target of `error()`”. Note that this change in temporal properties encodes both the change itself (from `isValidWidget()` and `isValidThread()` to `checkWidget()`) as well as the context (the `error()` method). Being able to express temporal ordering *as well as* data flow, and to include changes *as well as* context, is what makes this particular representation so well-suited to propagate changes at high precision.

2.1 Temporal Properties

We define software evolution as the evolution of temporal properties. But first, let us give details on how to extract temporal properties from a single project version.

Formally, a *temporal property* is an ordered pair of events a and b associated with the same object. We use the expression $a \prec b$ to represent an ordering where event a may happen before event b . An event associated with an object is one of the following:

- a method call (including constructor calls) with the object being used as the *target* or as an *argument*: `x.bar(y, z)` is an event associated with `x`, `y`, and `z`.
- a method call with the object being the value that was *returned* by the method: `x = map.items()` is an event associated with `x`.
- a field access with the object being the value that was *read*: `x = System.out` is an event associated with `x`.
- a cast with the object being *cast* to a different type: `(String) x` is an event associated with `x`.

Events are represented as precisely as possible (e.g., a method call is represented using the fully qualified name of the class defining the method, the method’s name and its signature). In the presented examples we omit most of those details to improve readability. We chose to focus on those types of events, as we found that these event types are well-suited to characterize the complex patterns of API usage in terms of their data flow and control flow [17]. The aim is to find a balance between the comprehensive results by static analysis and scalability.

To extract temporal properties we have adapted our earlier tool, JADET [17]. JADET works on bytecode level and extracts temporal properties in two steps:

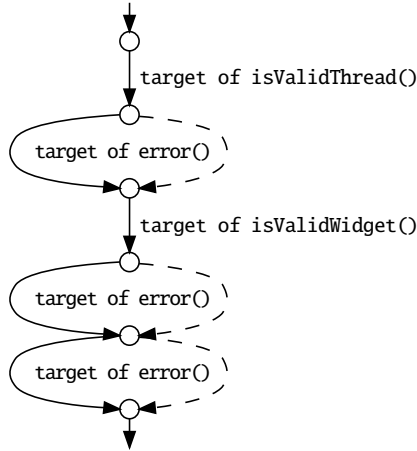


Fig. 3. Object usage model for the target object (“this”) accessed by the methods shown in Figure 1 (Eclipse 1.0 version). Dashed lines denote empty transitions (without calls).

Mining object usage models. Object usage models are finite state automata that show how objects “flow” through various events in a method.

Extracting temporal properties. Temporal properties, as extracted from object usage models, provide a succinct and easy-to-manipulate representation of how objects actually “flow” through various events.

To illustrate how JADET works, let us again consider the example method `removeSelectionListener()` from Eclipse 1.0 version in Figure 1. The first step is to mine object usage models. We create an object usage model for each statically identifiable object used by the method. These objects are: *formal parameters* of methods (including the implicit `this` parameter), *objects created* via `new`, *return values* of method calls (as in `x = map.items()`), values read from *fields* (including static fields, as in `x = System.out`), and explicit *constants* (such as `null` and `"OK"`). The `removeSelectionListener()` method uses three objects: the `listener` argument, the `eventTable` field, and the implicit `this` object. For each of those objects we build an object usage model that will show how the object is being used (i.e., in which events it participates). For example, the object usage model for the implicit `this` object is shown in Figure 3. Dashed edges represent “no-op” transitions. This model expresses the fact that the calls to `error()` are optional, whereas the two calls to `isValidThread()` and `isValidWidget()` always happen, and always in the same order².

² An astute reader will notice that the object usage model is not fully correct, because it assumes that after the call to `error()` the method’s execution proceeds further. This limitation is not too important, though, because such “bail-out” methods do not occur too often and thus their influence on the overall results is small.

After we have extracted object usage models, we can abstract each of them into a set of temporal properties. The idea here is as follows: If there is a path through the object usage model (from the initial state to any other state), along which events a and b occur, and there exists an occurrence of a that happens earlier on that path than some occurrence of b , we create a temporal property $a \prec b$ —expressing the fact that a may precede b . For this purpose we again use JADET, but with the following major modification. JADET, when abstracting object usage models into temporal properties, drops the dataflow information that is present in the models. For example, if an object usage model contains two successive events, “target of `lock()`” and “target of `unlock()`”, JADET will abstract it into a temporal property `lock() \prec unlock()`. For this work, we extended JADET to actually put *dataflow information* in the temporal properties. For example, extracting temporal properties from the object usage model shown in Figure 3 results in the following set:

```
target of Widget.isValidThread()  $\prec$  target of Widget.error()
target of Widget.isValidThread()  $\prec$  target of Widget.isValidWidget()
target of Widget.error()  $\prec$  target of Widget.isValidWidget()
target of Widget.error()  $\prec$  target of Widget.error()
target of Widget.isValidWidget()  $\prec$  target of Widget.error()
```

Once we do this for every single object usage model extracted from a method, we can create a union of those models’ temporal properties and store it as the set of temporal properties that characterize the method. If we consider the method `removeSelectionListener()` from Figure 1 (Eclipse 1.0 version), the set of temporal properties that characterizes it will contain the temporal properties shown above and the following temporal properties, obtained from the object usage model created for the `listener` argument and the `eventTable` field (the other two objects accessed by `removeSelectionListener()`):

```
2nd arg of EventTable.unhook()  $\prec$  2nd arg of EventTable.unhook()
field Widget.eventTable  $\prec$  target of EventTable.unhook()
```

Further details on extracting temporal properties can be found in the paper by Wasylkowski et al. [17].

2.2 Change Properties

Temporal properties tell us how a specific method uses APIs. Therefore, if we want to see how the APIs usages in a method evolved and changed between two versions of a project, we can look at how the temporal properties of the method changed. For this purpose, we introduce the notion of *change properties*, constructed from the comparison of two sets of temporal properties coming from two versions of the same method. To identify the same method between two versions of the project, we use the method’s name, signature, and the class in which it is defined (if a method was renamed, we will not be able to track its evolution). A change property is a temporal property annotated with information about the temporal property’s evolution between the two versions (more on that

below). We built LAMARCK for the purpose of extracting change properties. LAMARCK extracts change properties in three stages, which we detail using the example from Figure 11.

Extracting Temporal Properties. In the first stage, LAMARCK identifies the common methods between the two versions of the analyzed project and extracts the sets of *temporal properties* for each method in each version separately.

As an example, consider versions 1.0 and 2.0 of Eclipse. One of the methods that occurs in both versions is `removeSelectionListener()` from the `Button` class, shown in Figure 11. After identifying all common methods between the two versions, LAMARCK extracts temporal properties for each method in each version with the help of the modified JADET tool (as explained in Section 2.1). Here is the set of temporal properties for the `removeSelectionListener()` method, as implemented in Eclipse 1.0³:

```
EventTable.unhook() < EventTable.unhook()
field Widget.eventTable < EventTable.unhook()
Widget.error() < Widget.error()
Widget.error() < Widget.isValidWidget()
Widget.isValidThread() < Widget.error()
Widget.isValidThread() < Widget.isValidWidget()
Widget.isValidWidget() < Widget.error()
```

The temporal properties in version 2.0 are much simpler:

```
EventTable.unhook() < EventTable.unhook()
field Widget.eventTable < EventTable.unhook()
Widget.checkWidget() < Widget.error()
```

Extracting Change Properties. In the second stage, LAMARCK compares the sets of temporal properties in both versions of each method and creates a set of *change properties* for this method—that is, temporal properties annotated with information about their evolution. If a temporal property is only present in the first version, LAMARCK transforms it into a change property annotated with *D* (for *deleted*); if a temporal property is only present in the second version, LAMARCK transforms it into a change property annotated with *A* (for *added*). In our `removeSelectionListener()` example, LAMARCK will create the following set of change properties:

```
D: Widget.error() < Widget.error()
D: Widget.error() < Widget.isValidWidget()
D: Widget.isValidThread() < Widget.error()
D: Widget.isValidThread() < Widget.isValidWidget()
D: Widget.isValidWidget() < Widget.error()
A: Widget.checkWidget() < Widget.error()
```

³ From now on, we will omit presenting dataflow information in temporal properties, except if it is really needed to understand the property, in order to increase the readability of the properties in the paper.

Table 1. Change properties (including context) for the `removeSelectionListener()` method

```

O: EventTable.unhook() < EventTable.unhook()
O: field Widget.eventTable < EventTable.unhook()
O: Widget.error() < Widget.error()
O: Widget.error() < Widget.isValidWidget()
O: Widget.isValidThread() < Widget.error()
O: Widget.isValidThread() < Widget.isValidWidget()
O: Widget.isValidWidget() < Widget.error()
D: Widget.error() < Widget.error()
D: Widget.error() < Widget.isValidWidget()
D: Widget.isValidThread() < Widget.error()
D: Widget.isValidThread() < Widget.isValidWidget()
D: Widget.isValidWidget() < Widget.error()
A: Widget.checkWidget() < Widget.error()

```

These change properties show how temporal properties of the example method `removeSelectionListener()` evolved between versions 1.0 and 2.0 of Eclipse.

Adding Context. In the final stage, LAMARCK extends the set of change properties created for each method in the previous stage with special change properties expressing the *context* of the change. For this purpose, LAMARCK adds to the set of change properties created in the previous stage another set of change properties, obtained by annotating *all* temporal properties from the earlier version of the method with *O* (for *original*)—these properties we consider the context of a change. Whereas the change properties created in the second stage represent the change itself—and thus will allow us to find *evolution patterns*, the change properties created in this stage represent the *context* of the change—and will allow us to find locations in the project where the change should have happened, but did not. This will allow us to find *missing changes*—the main contribution of this paper.

In our example, Table 1 shows the final set of change properties extracted by LAMARCK for the example method from Figure 1.

In a similar manner, LAMARCK goes through all the methods common to two versions of the analyzed project and produces a set of change properties for each such method.

3 Mining Patterns

Having generated the change properties, LAMARCK can use them to mine evolution patterns (i.e., sets of change properties that repeat in many methods) and find missing changes (i.e., methods where a certain change should have been applied but was not).

3.1 Detecting Evolution Patterns

In Section 2, we have shown how LAMARCK can express evolution of methods using their change properties. Because we are interested in tracking the evolution of the whole project, we need to *aggregate* the individual changes over the entire project. More specifically, if a certain set of change properties occurs frequently throughout the project’s evolution (i.e., is common to many methods), we treat it as an *evolution pattern*. For detecting evolution patterns LAMARCK uses *formal concept analysis* and its implementation provided by the Colibri/Java tool [6].

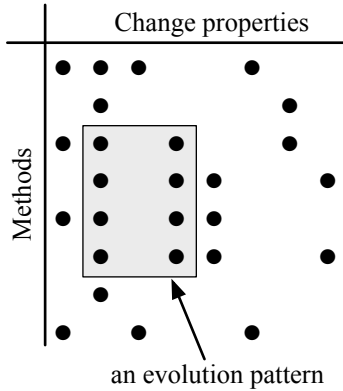


Fig. 4. Concept Analysis Matrix

Formal concept analysis is, broadly speaking, a technique for finding patterns [5]. Its input is a set of objects, a set of properties, and a cross table associating objects with properties. In our case, the set of objects is the set of common methods between the two analyzed versions. The set of properties is the set of all change properties created from the entire project. Figure 4 shows an example of such a cross table. A dot is present in a place where a row and a column cross when the change property represented by the column was extracted for the method represented by the row. The result of formal concept analysis are *concepts* (rectangles) found in the cross table. A concept is a set of objects and a set of properties such that every object in the concept is associated with all properties in the concept, and both sets are maximal (i.e., it is not possible to add elements to one of the sets without influencing the other one). In our case, a concept is a set of change properties and a set of methods such that the change properties occur in each method. The number of methods in the concept is called the *support* of the concept.

If we restrict ourselves to finding concepts that have high support values, we will in effect be finding sets of change properties that occur in many methods—and therefore likely candidates for *evolution patterns*. For this purpose we use a parameter called the *minimum support*. LAMARCK returns only those concepts that have support values at least equal to the value of the minimum support parameter, and treats all returned concepts as evolution patterns.

Table 2. An evolution pattern occurring in 170 Eclipse methods

```

O: Widget.error() < Widget.error()
O: Widget.error() < Widget.isValidWidget()
O: Widget.isValidThread() < Widget.error()
O: Widget.isValidThread() < Widget.isValidWidget()
O: Widget.isValidWidget() < Widget.error()
D: Widget.error() < Widget.error()
D: Widget.error() < Widget.isValidWidget()
D: Widget.isValidThread() < Widget.error()
D: Widget.isValidThread() < Widget.isValidWidget()
D: Widget.isValidWidget() < Widget.error()
A: Widget.checkWidget() < Widget.error()

```

Table 2 shows an example of an evolution pattern as extracted by LAMARCK from Eclipse versions 1.0 and 2.0. This evolution pattern expresses the change to the `removeSelectionListener()` method shown in Figure 1. As it was applied to 169 other methods in Eclipse, the support of this evolution pattern is 170.

3.2 Finding Missing Changes

Finding evolution patterns is useful for understanding and for documentation purposes, and in Section 4.1 we give examples of interesting and useful evolution patterns found by LAMARCK. However, there is a very important question that can often occur while changing a project: *Was the change applied consistently throughout the entire project's code?* If we consider the evolution pattern shown in the preceding section, methods where the change expressed by the pattern was not applied, but should have been, become locations with potential future defects and/or maintenance problems. Therefore, it is important to be able to answer the stated question.

Our evolution patterns contain, amongst others, change properties annotated with the letter “O”—these are temporal properties that were present in the earlier version. They form the *context* of the change, and answer the question: *In which context does the change happen?*

As an example of such a change context, consider the evolution pattern shown in the preceding section, Table 2. Any method that has in its change property set all the change properties annotated with “O” from this pattern, but none of the other “A” or “D” change properties, is in fact a method that exhibited the same “starting conditions”, like other methods that evolved, but it itself did not evolve. Potentially, there is a missing change in that method.

LAMARCK finds such missing changes again using the Colibri/Java tool [6] for formal concept analysis. Generally, whenever we have two concepts, and the set of properties in one of them is a subset of the set of properties in the other one, we have a potential violation. The reason why one set is a subset of another is that there are some properties missing—in our case, missing change properties. For details on the technique, see [11].

However, it can happen that an evolution pattern occurs in many methods, but is also missing in many methods. As it is not possible to say with absolute certainty that all those other methods are violating the pattern, we introduce another parameter—the so-called *minimum confidence*. The *confidence* of a violation is a number between 0 and 1, calculated as the ratio $s/(s+v)$, where s is the support of the evolution pattern (i.e., the number of methods that did evolve exactly according to the pattern), and v is the number of methods that violate the pattern (i.e., those that exhibit only the “O”-annotated change properties from the pattern).

4 Evaluation

LAMARCK detects evolution patterns that carry information regarding which sequences of method calls have been removed or added in a certain method between two versions of the project. In this section, we are going to evaluate LAMARCK’s usefulness.

We evaluate LAMARCK in two different evaluation settings:

Detecting errors. In Section 4.1, we run LAMARCK on the subjects “as is” and manually assess the reported violations for whether they are real code issues. It turns out that 33%–62% of the reported violations indeed are code smells or defects. In this scenario LAMARCK looks for missing changes, i.e. locations that were omitted by the developers when applying a change.

Preventing errors. In Section 4.2, we simulate settings in which programmers had to apply a change and we wanted to see if LAMARCK can assist them during this process and prevent them from omitting a change. It turns out that in such a setting, LAMARCK has almost no false alarms (the precision ranges from 90%–100%); almost all of the inconsistencies detected by LAMARCK actually were in need for update and later versions this update was performed in exactly the way as predicted by the pattern.

For our experiments, we used the projects and versions⁴ given in the first column in Table 3. In the second and third columns we give the number of methods in the earlier and the later versions of the analyzed project. The fourth column contains the total number of methods that we managed to match between the two project versions, expressed both as an absolute value and a percentage of the number of methods in the earlier version. All of the presented results in the evaluation section are computed for minimum support value of 10 and minimum confidence value of 0.8.

4.1 Detecting Errors

In the first part of our evaluation, we want to assess how well LAMARCK is able to detect defects due to missing changes. For this purpose, we applied it to the

⁴ <http://archive.eclipse.org/eclipse/downloads/>, <http://www.eclipse.org/aspectj>,
<http://sourceforge.net/projects/azureus/develop>

Table 3. Evaluation Subjects

Case study	# Methods			#Patterns	Time
	older version	newer version	Matching		
Eclipse 1.0 vs. 2.0	34,197	49,862	19,952 (58%)	133	13m34s
Eclipse 2.0 vs. 2.1	49,862	61,358	44,692 (90%)	2,019	10m57s
AspectJ 1.6.0 vs. 1.6.3	38,544	36,729	35,546 (92%)	58	7m41s
Azureus 4.1.0.0 vs. 4.4.0.0	38,328	40,506	32,200 (84%)	17	9m57s

subjects described in Table 3; if a violation of a pattern is reported, this means that a location in the project’s code has been found that does not comply with the change pattern extracted from the project’s history. In other words, we have found a location where the developers were supposed to change something, but they did not and thus we find a missing change.

Hypothesis 1: LAMARCK detects missing changes.

Patterns reported. We ran LAMARCK on the test subjects presented in Table 3. In the last two columns of Table 3, we give the number of extracted evolution patterns and the total analysis time in minutes and seconds (including the time used to extract object usage models and temporal properties). The way to interpret the number of patterns detected is generally “the more, the better”, as the more patterns we detect the more errors we will be able to catch. The time results are for an Intel Core 2 Duo 2.57 GHz machine with 4GB of RAM, averaged over 10 runs.

The first thing that stands out is the big difference in the number of patterns detected. This is due to the size of the projects and the difference between the versions. Generally speaking, if the two versions are too far apart in time, the source code would have evolved too much for us to manage to match the methods. This becomes evident in the change from Eclipse 1.0 to Eclipse 2.0. Here, only 133 patterns are reported; as Eclipse has changed quite a lot for this initial period there were only a few common methods (see Table 3) that could be detected and only so much patterns reported. In contrast, consider the 2019 reported patterns for Eclipse 2.0 vs. 2.1; we attribute this much higher number to the closeness of the structure of Eclipse in this minor release increment.

One would also notice the difference in the patterns detected in Eclipse and in the Azureus and AspectJ projects. This is due to the fact that the latter two are smaller than Eclipse, leading to fewer changes in fewer locations. We chose a minimum support of 10 for all of our experiments, which means that a pattern should appear in at least 10 locations, which is not so often the case for smaller projects.

We present a few examples of interesting patterns later in this section.

Issues detected. Using the detected evolution patterns, LAMARCK looks for possible violations of those patterns in order to detect missing changes in the

Table 4. Reported unique violations and their success rate

Case study	Time	#Violations	%Issues
Eclipse 1.0 vs. 2.0	62s	6	33%
Eclipse 2.0 vs. 2.1	111s	8	62%
AspectJ 1.6.0 vs. 1.6.3	72s	7	57%
Azureus 4.1.0.0 vs. 4.4.0.0	86s	5	40%

project’s code. A violation reported by LAMARCK need not be an issue. We manually investigated all of the reported violations and classified them into the three categories of code defects, code smells (i.e., potential defects) and false positives.

We define code defects, code smells and false positives as follows:

- **Defects.** Those are real defects in the source code that will lead the project to fail by crashing or producing erroneous results.
- **Code smells.** This category contains all reported violations that are not at present defects in the code, but have the potential to become such. In this category fall also all reported cases, which might be improved in terms of readability, maintainability or performance of the program.
- **False positives.** This category contains all reported violations, that are neither defects, nor code smells.

In Table 4 we report our findings. The first column of Table 4 lists again our test subjects. The second column gives the time needed in seconds for LAMARCK to detect the patterns violations for each of our case study subjects. The third column contains the number of unique patterns violations detected by LAMARCK. The success rate of the reported violations is presented in the last column of the table and takes into account both the reported defects and code smells (summarized as “issues”), as in both cases the source code needs to be corrected. Our highest true positive rate is 62%—that is, 62% of the locations where LAMARCK detected an issue were in need of correction. Even though a false positive rate of 38% does indicate that there is still room for improvement, our results show that every second of our reports points to a bug and this is in production code, which should have far fewer defects. This result highlights the potential benefits of our approach.

Two of our subjects had much lower true positive rates:

- Between Eclipse 1.0 and 2.0, the true positives rate is 33%—that is, 2 out of 3 violations are false alarms. This is due to several refactorings between these two major releases and most of the reported false positive pattern violations were reported for methods that have evolved too much for the pattern to still hold. In other words, the distance is too large to learn from.
- Most of the false positives reported by LAMARCK for the Azureus project were due to the different compiler versions used for compiling the two versions. For our experiments we used the binaries provided by the project and

```

void createControl(Composite parent) {
+ Font font = parent.getFont();
  Composite comp = new Composite();
  ...
  Composite locationComp = new Composite();
  GridLayout locationLayout = new GridLayout();
  ...
  locationComp.setLayout(locationLayout);
  GridData gd = new GridData(GridData.FILL_BOTH);
  locationComp.setLayoutData(gd);
+ locationComp.setFont(font);
  ...
}

```

Fig. 5. Method change from Eclipse 2.0 to 2.1. The added methods address a font inconsistency.

as LAMARCK is working on binary level, this resulted in reporting of violations that boil down to equal source code, but different binary code. One should note that these are not faulty recommendations as the byte code did indeed change, but this change just had no visible manifestation in the end source code.

Generally speaking, any static defect detection tool will suffer from false positives—in particular, if the properties checked against were learned from other code instances. The question is whether the issues could also be detected in another, possibly cheaper way. Tools like FindBugs⁵, for example, check for specific API misuses by implementing a specific analysis for each API. This would be cheaper (in terms of computational power) and more precise, yet less general and more expensive (in terms of human labor). For the kind of issues LAMARCK detects, there is yet no other alternative; and the our results indicate a reasonable efficiency.

Qualitative Analysis. Now let us take a look at a few examples of the patterns and their violations detected by LAMARCK.

Our first example is the one presented in Figure 5. As one can see from this source code example new method calls have been added to the method. This change is detected by LAMARCK by the following pattern:

```

O: Composite.<init>() < Control.setLayout()
O: Composite.<init>() < Control.setLayoutData()
A: retval of Control.getFont() < 1st arg of Control.setFont()

```

This pattern tells us that when we are setting the layout and the layout data on a `Control` object, we should also set the font on the same object (the `Composite` class inherits from the `Control` class). It originates from a widely spread issue

⁵ <http://findbugs.sourceforge.net/>

```

void navigation(IProgramElement node)
{
    if (node == null) return;
    ...
-   treeViewBuilder.buildView
    (Asm.getDef().getHierarchy());
+   treeViewBuilder.buildView
    (Ajde.getDef().getModel().getHierarchy());
}
...
}

```

Fig. 6. Method change from AspectJ. `buildView()` now takes an object derived from the Ajde singleton.

```

public AddBookmarkAction(Shell shell)
{
    super(WorkbenchMessages.getString("AddBookmarkLabel"));
    setId(ID);
    Assert.isNotNull(shell);
    this.shell = shell;
    setToolTipText(WorkbenchMessages.getString("AddBookmarkToolTip"));
-   WorkbenchHelp.setHelp(this, new Object[] {IHelpContextIds.ADD_BOOKMARK});
+   WorkbenchHelp.setHelp(this, IHelpContextIds.ADD_BOOKMARK);
}

```

Fig. 7. Method change from Eclipse 1.0 to 2.0. The changed method `AddBookmarkAction` calls a deprecated method.

in Eclipse 2.0, where people were not setting the font of the `Control` object they were working with, which could lead to problems when trying to set a font in a control based on the font of its parent. In Eclipse 2.1 this issue was corrected by adding calls to `getFont()` and `setFont()` in the appropriate locations, resulting in LAMARCK detecting it as a pattern. The support value for this pattern is 66, i.e. it was applied 66 times during the change from Eclipse 2.0 to 2.1. The violation of the pattern expresses itself in missing the following change property:

A: retval of `Control.getFont()` \prec 1st arg of `Control.setFont()`

what this tells us is that there was supposed to be added a call to `getFont()` before the call to `setFont()`. LAMARCK was able to detect locations in the Eclipse 2.1 code, where this change was not applied, thus marking those locations as locations that violate an evolution pattern and expose a code defect in the project. What this means is that in the faulty methods the developer is creating a new component, but is not setting its font properly. As discussed earlier the omission of the `getFont()` method call was a widely spread issue in Eclipse 2.0. The violation found by LAMARCK shows that the Eclipse developers failed to locate all methods where this change needed to be applied. In this case LAMARCK was able to point to such omitted locations.

Our second example comes from the AspectJ project and is shown in Figure 6. The change pattern that LAMARCK detected for this piece of code is the following:

```
O: retval of Asm.getDef() < Asm.getHierarchy()
D: retval of Asm.getDef() < Asm.getHierarchy()
A: retval of Ajde.getDef() < Ajde.getModel()
A: retval of Ajde.getModel() < Asm.getHierarchy()
```

What this pattern, with a support of 10, tells us is that the return value of the deleted `getDef()` method call was substituted with the return value of the added `getDef()` and `getModel()` method calls. As one can notice, the deleted methods are from class `Asm`, while the added ones are from class `Ajde`. After investigating the case, it turns out that the `Asm` class was a singleton class, but it was changed in the newer AspectJ version to a non-singleton class, due to change in the AspectJ functionality. A new class `Ajde` was created that acted as a singleton wrapper of the old class and had a `Asm` field (that is why the `Asm.getHierarchy()` is called on the `Ajde.getModel()` return value). In the newer version of AspectJ both classes are present (and all their methods, as well). Thus, a developer could still use the `Asm.getDef()` method and the compiler would not issue a warning. However in this case, the `Ajde` class needs to be used.

Now let us take a look at an example of a code smell detected by LAMARCK. In Figure 7 one can see a frequently occurring change between Eclipse 1.0 and 2.0. This change has been detected by LAMARCK through the following pattern:

```
O: setToolTipText(String) < setHelp(iAction, Object[])
D: setToolTipText(String) < setHelp(iAction, Object[])
A: setToolTipText(String) < setHelp(iAction, String)
```

What this pattern tells us is that the call to `setHelp(iAction, Object[])` has been deleted and substituted with a call to `setHelp(iAction, String)`. The support for this pattern is 33, which means that in 33 methods throughout Eclipse 2.0 this pattern has been followed. LAMARCK has detected violations of this pattern in Eclipse 2.0 in the sense that the developers continued to use the old `setHelp` method. After investigating the matter we found out that the old `setHelp` method was in fact a deprecated method in Eclipse 2.0 and shouldn't have been used. Thus, even though the program was not crashing, we classified this violation as a code smell, as the code is expressing unwanted and deprecated behavior.

In conclusion we can state that

LAMARCK is able to find useful evolution patterns and use them to detect missing changes in project code.

4.2 Preventing Errors

We already discussed a few of the patterns detected by LAMARCK and their usefulness when it comes to detecting missing changes in project's code. However,

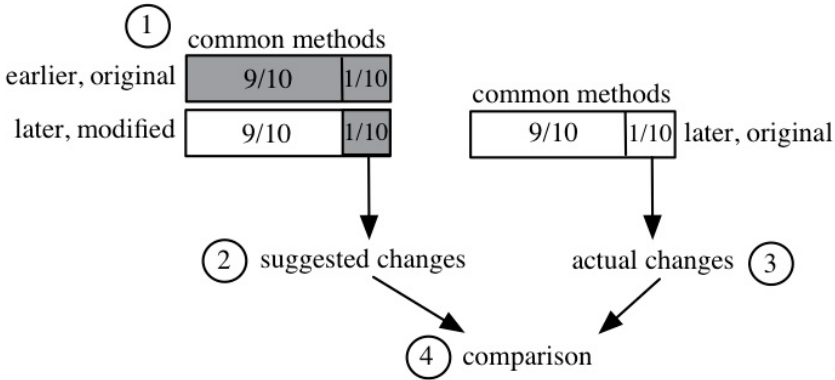


Fig. 8. Evaluation Setting. In 1/10 of the code, we artificially revert changes and check whether LAMARCK is able to predict them after learning from the changes in the remaining 9/10.

this is only one of the aspects of the usefulness of those patterns. The patterns as detected by LAMARCK can also be used for *preventing* errors. On top of that, the patterns themselves are an exact suggestion how to fix a potential future defect location.

Hypothesis 2: LAMARCK prevents missing changes and suggests how to fix them.

Evaluation Setting. In order to perform such evaluation in an unbiased manner, we designed the scenario sketched in Figure 8. The key idea is to *artificially revert changes* between versions and check whether LAMARCK is able to predict them. In this way we simulate a real-life scenario, when a given change pattern has been applied to only a few of the intended locations.

- ① We split the set of common methods in the two versions into two parts—9/10 and 1/10 parts. In the 1/10 part, we substituted the methods from the later version by the methods in the earlier version, effectively reversing the changes that occurred between those two versions. We thus simulated a situation in which 1/10 of the code in the later version would still be in need of update.
- ② We applied LAMARCK on the earlier version and the (modified) later version and had LAMARCK predict which locations in those 1/10 methods would be in need for update and what the update should accomplish.
- ③ From the (original) later version, we looked at the *actual changes* applied in the same 1/10 methods subset.
- ④ Comparing the *suggested* and the *actual changes* for the 1/10 part allows us to assess the accuracy (and hence the usefulness) of LAMARCK.

Using the setting described above, we performed 50 random 1/10 vs. 9/10 splits on each of the four case studies.

Table 5. LAMARCK’s effectiveness in discovering inconsistently applied changes. The table summarizes the results obtained for 50 random splits of the input data. See Section 4 for details on the evaluation scheme.

Case study	# Inconsistencies			Precision		
	min	max	avg	min	max	avg
Eclipse 1.0 vs. 2.0	16	29	22	93%	100%	98%
Eclipse 2.0 vs. 2.1	14	24	19	90%	100%	99%
AspectJ 1.6.0 vs. 1.6.3	4	12	7	100%	100%	100%
Azureus 4.1.0.0 vs. 4.4.0.0	2	5	3	100%	100%	100%

Results. Our results are presented in Table 5. The first column gives the projects and their versions used for the experiment. The second column gives the number of inconsistencies (with a pattern) detected in the 1/10 part of the methods, when the later version was modified as explained above—we show the minimum, maximum and the average number of reported inconsistencies over all the 50 experimental runs. The third column contains the *precision* of the reported inconsistencies, i.e. the percentage of the cases where the faulty location was indeed fixed by the developers exactly as LAMARCK recommended it to be fixed. A high precision implies a low number of false positives, i.e. invalid recommendations.

The reported precision in Table 5 is accumulated over the total number of 50 random splits. As one can see from the table, our precision results are close to 100% in all cases. This means that almost all of the inconsistencies detected by LAMARCK actually were in need for update, and this in the exact way as predicted by the pattern. We consider this precision a remarkable result, as this means LAMARCK is not only able to predict that some location will change, but is able to accurately predict how this location will change. Please note that these results do not depend on the size of the split (if we consider all the changes applied in 90% of the code or less), as the size of the split would influence the amount of patterns detected, but not their defect prevention properties.

Validation. In our experience, evaluation results like these are more likely to indicate a bug rather than a feature. We therefore manually took a look at one random split for each of the four experiments in order to re-verify that the location for which an inconsistency would have been reported was indeed changed the way our patterns say. Our findings were that in all cases what LAMARCK predicted that has to be changed, the code was indeed changed and that exactly in the manner predicted. We classified predictions that were wrong as false positives.

As LAMARCK extracts the evolution patterns from the bytecode of the project versions, we also stumbled across evaluation artifacts—cases where our tool recommends a change, but this change is only in the type of the objects returned by some methods and passed to some other methods. The situation here is as follows: class *A* gets replaced by class *B*, and methods that used class *A* now

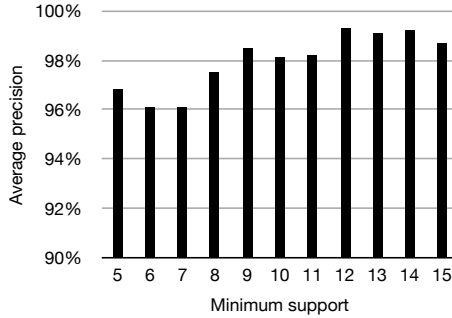


Fig. 9. Influence of minimum support on LAMARCK’s effectiveness on Eclipse 1.0 vs. 2.0 (minimum confidence fixed at 0.8)

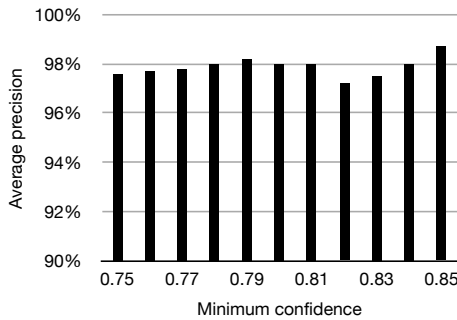


Fig. 10. Influence of minimum confidence on LAMARCK’s effectiveness on Eclipse 1.0 vs. 2.0 (minimum support fixed at 10)

use class B . Source code that does not use objects of class A explicitly, but just passes them around (as in `foo(bar())`, if `bar()` returns an object of type A and `foo()` accepts an object of type A) does not need to be changed, but the bytecode will change after the project gets recompiled. Since in our evaluation we use old bytecode of the 1/10 part, LAMARCK reports such locations as needing an update. However, these are not faulty recommendations *per se*, because during real usage (and not in an artificial setting, as in our evaluation) after replacing classes and recompiling the code LAMARCK will not report such locations anymore.

After classifying all reported predictions into true positives, false positives and evaluation artifacts, our manual inspection reports precision ranging from 89% to 100%. We classified in total 45 predictions out of which 16 were evaluation artifacts, which we ignored. Thus, our manual inspection confirmed our high precision rate.

To further reduce the probability of a bug, we also took a look at the *recall* values and they were as close as up to 10%. Recall in this case would mean finding all locations where a change is supposed to be applied. Such low recall values are expected, as we do not aim at finding all changes that occurred between two versions of a given project; we aim at finding all *frequently* occurring changes—changes that follow certain patterns. The recall values are low, as most of the

code changes occur only at a single location (or at least in less than 10 locations, which is our minimum support number). A bug in the evaluation, for instance a mix of training and testing sets, would have returned far higher recall values.

Sensitivity Analysis. Finally, we investigated the sensitivity of our results to small changes of the minimum support and minimum confidence parameters. For this purpose, we redid our evaluation on Eclipse 1.0 vs. 2.0 for different minimum support and minimum confidence values. The results are shown in Figures 9 and 10. It turns out that LAMARCK is quite insensitive to small changes of its input parameters. What is more important, for minimum support of 5, LAMARCK finds on average 40 missing changes (compared to 22 when using the default minimum support value of 10), and still more than 95% of LAMARCK’s suggestions (on average) are followed by the Eclipse developers. Thus, users can tweak LAMARCK to find more missing changes, and still get very precise results.

All these results confirm our hypothesis:

*LAMARCK can detect missing updates with a precision close to 100%,
giving precise fix suggestions.*

4.3 Threats to Validity

As any other, our study is prone to threats to validity.

External validity. We have investigated seven versions of three different open-source projects, coming from different maturity, size and domain. However it is possible that the results we acquire on them do not generalize to other arbitrary projects. For example, closed-source projects, due to differences in the internal processes, might have very different properties.

Construct validity. Our approach might be prone to mistakes. The external tools we use might also be defective. However, we hope that we have eliminated this threat to a big extent as the Colibri [11] and the JADET [17] tools are publicly available⁶ and besides the validation in Section 4.2, we ourselves have performed a cross-check of our source code to eliminate any possible mistakes on our side.

Internal validity. The presented evaluation for the usefulness of LAMARCK when used as an errors preventing tool is a combination of automatic and manual inspection of 50 random splits of the common methods for two versions of a project methods. It might be the case that 50 splits are not enough. It might also be the case that, as we are not well acquainted with the analyzed projects, our manual classification results would not be the same if classified by a real developer from that project. Another possibility is that, apart from the evaluation artifacts mentioned in Section 4.1, other missing

⁶ This is the online implementation of the latest JADET tool version: <http://www.checkmycode.org/>

changes reported by LAMARCK would also be found by a compiler (due to missing types, etc.). However, based on our evaluation in Section 4.2, we think that the likelihood of *all* reported missing changes being of this type is very small.

5 Related Work

Our approach is unique in that it combines specification mining with mining source code archives. LAMARCK is based on the JADET [17] static analysis tool, but the same technique could be used to enrich any other single-versioned programming rules mining tool like PR-Miner [10] or GrouMiner [16].

To the best of our knowledge, the presented work is the first to define API evolution patterns as a set of change properties derived from temporal properties. However, there are many other approaches that learn from existing code in order to learn about the software evolution or detect code defects.

5.1 Learning Evolution Rules

A large body of work has been done in the area of looking for API evolution changes and the way they should be deployed to API clients. Several techniques and tools [9,2,18,20] have been developed to discover the refactorings that a software system has undergone by analyzing two versions of the evolved software project. Dig and Johnson [3] found out that 84%–97% of all API breaking changes, i.e. changes that are extremely disruptive in the development life cycle of component-based applications, are in fact refactorings (e.g. class or method renaming). LAMARCK is also able to detect change patterns based on refactorings, but is also as well able to detect much more complicated patterns and thus find non-refactoring based defects.

Nguyen et al. [14] developed the LibSync tool, which helps developers migrate from one library version to another. LibSync has a knowledge base of API adaptation patterns for each library version and given a client system and the desired library version, the tool finds the locations in the code that are associated with the changed API version. In comparison, even though both tools report similar precision rates, LAMARCK is much more light-weight and time-efficient and is able to detect both external API, as well as project-specific change patterns.

Based on how a framework adapts to its own changes, Dagenais and Robillard [1] developed a recommendation system that suggests replacements for framework elements accessed by client programs. Dagenais and Robillard designed a tool, called SemDiff, that explores code locations that used an API method, which was later deleted from the API. SemDiff mines the new API methods that are being used instead and comes up with a set of method calls that substituted the call to the deleted API method. In comparison to SemDiff, LAMARCK is also able to perform such kind of detection, but in addition can also offer recommendations in the cases where an entire piece of code was substituted with a method call (e.g. our `checkWidget()` example) or the method

(or even a class) remained unchanged, but the usage pattern of the method (or the class) changed (e.g. our `Ajde` example).

Fluri et al. [4] looked for context changes of method invocations as moving an existing method invocation into the `then` or the `else`-part of an `if`-statement. Our approach is not restricted to such context changes and can detect any type of context change thanks to the *original* change properties representing the context of a change.

Kim and Notkin [7] grouped code changes that form change patterns with the help of their `LSdiff` tool. `LSdiff` infers systematic structural code differences as logic rules from the difference of two sets of predicates, representing the two versions of a program. `LSdiff` was built as a tool for assisting developers when making a `diff` between two revisions of a file.

5.2 Learning from Project History

`FixWizard` [15] is a tool that identifies recurring bug fixes by comparing the changes that happened between two version control revisions of a project. Apart from using a completely different algorithm for identifying frequently occurring changes, `LAMARCK` and `FixWizard` also interpret the context of a code change differently. `FixWizard` is restricted to infer and offer recommendations only from and to code peers that match in naming convention or ancestor classes. `LAMARCK` on the other hand interprets context as any method that meets the same “starting conditions” (described by the original temporal properties) of a pattern, thus addressing a much larger number of methods.

Livshits and Zimmermann [12] also mine patterns and their violations from software repositories. Their `DynaMine` tool can detect a pattern of method calls, but only if the method calls are used in the same transaction. We on the other hand, look at the project as a whole and extract our patterns based on the entire project’s code. This allows us to produce more general patterns and to find patterns and possible defect locations different from the ones `DynaMine` detects.

Kim et al. [8], similarly to Livshits and Zimmermann [12], also operate on version system transaction level and look for patterns on bug fixes. Our approach can, like `BugMem`, detect changes that were due to a bug fix, but is not restricted to that. For example, we are able to detect a change due to the addition of new code, which is not the case with `BugMem`. We also have a much higher true positive rate, when comparing our best defect detection results (62%) against their best results (38.7%) on the Eclipse project (see Section 4.1).

Williams and Hollingsworth [19] mine source code repositories to look for commonly fixed bugs. We do not rely on version repositories, which are not always easy to find, but simply on two versions of the same project. Our method is not restricted to patterns of bug fixes.

6 Conclusion and Consequences

To reduce the risk induced by software evolution, it is necessary that changes be applied *consistently* across a project. By characterizing the impact of change

on involved method calls, their temporal ordering, and their dataflow, our tool LAMARCK learns how software has changed in the past. As it comes to *preventing* errors, LAMARCK's recommendations are very precise. An average false positive rate of $< 2\%$ implies substantial benefits at low costs and low requirements. We therefore recommend usage of LAMARCK or similar approaches in all projects that care about minimizing the risks of inconsistent software evolution. On top of that, LAMARCK can also be helpful for *detecting* errors in existing code, uncovering complex API usage changes with a true positive rate of 33%–62%.

Despite these successes, there is still a lot of work to do. Besides general improvements in performance and scalability, our future work will concentrate on the following topics:

Generating changes. The patterns LAMARCK produces contain all the information to deploy the change again and again. We are working on techniques that automatically generate a change to source code which only needs to be acknowledged by the programmer.

Automatic change deployment. Given the high precision of its recommendations, it may be possible to not only suggest a code change, but also to actually apply it. Such automatic updates, backed by additional regression tests or symbolic verification, may be useful in situations where loss of functionality is to be avoided under all circumstances.

Clone detection. Since object usage patterns serve so well in characterizing changes as well as context, we are currently investigating whether they would prove helpful tools in detecting code clones—an area where light-weight semantic approaches may meet a sweet spot between precision and applicability.

We made parts of our data and evaluation results available online. To learn more about our work in model mining, visit our Web site:

<http://www.st.cs.uni-saarland.de/models/>

Acknowledgments. Yana Mileva is funded by Microsoft Research and Max Planck Institut für Informatik. Gordon Fraser, David Schuler and Kim Herzig provided helpful comments on earlier drafts of this paper. The authors thank the anonymous reviewers for their feedback.

References

1. Dagenais, B., Robillard, M.P.: Recommending adaptive changes for framework evolution. In: ICSE 2008: Proceedings of the 30th International Conference on Software Engineering, pp. 481–490. ACM, New York (2008)
2. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated detection of refactorings in evolving components. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 404–428. Springer, Heidelberg (2006)

3. Dig, D., Johnson, R.: The role of refactorings in API evolution. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 389–398. IEEE Computer Society, Washington, DC, USA (2005)
4. Fluri, B., Zuberbühler, J., Gall, H.C.: Recommending method invocation context changes. In: RSSE 2008: Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering, pp. 1–5. ACM, New York (2008)
5. Ganter, B., Wille, R.: Formal concept analysis: Mathematical foundations. Springer, Heidelberg (1999)
6. Götzmann, D. N.: Formale Begriffsanalyse in Java: Entwurf und Implementierung effizienter Algorithmen. Bachelor thesis, Saarland University, Publication and software available from (2007), <http://code.google.com/p/colibri-java/> (accessed April 6, 2010)
7. Kim, M., Notkin, D.: Discovering and representing systematic code changes. In: ICSE 2009: Proceedings of the 31st International Conference on Software Engineering, pp. 309–319. IEEE Computer Society, Washington, DC, USA (2009)
8. Kim, S., Pan, K., James Whitehead, J.E.E.: Memories of bug fixes. In: SIGSOFT 2006/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 35–45. ACM, New York (2006)
9. Kim, S., Pan, K., Whitehead Jr., E.J.: When functions change their names: Automatic detection of origin relationships. In: Proceedings of the 12th Working Conference on Reverse Engineering, pp. 143–152. IEEE Computer Society, Washington, DC, USA (2005)
10. Li, Z., Zhou, Y.: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 306–315. ACM, New York (2005)
11. Lindig, C.: Mining patterns and violations using concept analysis. Technical report, Saarland University, Saarbrücken, Germany (June 2007)
12. Livshits, V.B., Zimmermann, T.: Dynamine: Finding common error patterns by mining software revision histories. In: Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 296–305. ACM, New York (2005)
13. McKean, E. (ed.): The New Oxford American Dictionary, 2nd edn. Oxford University Press, Oxford (2005)
14. Nguyen, H.A., Nguyen, T.T., Wilson Jr., G., Nguyen, A.T., Kim, M., Nguyen, T.N.: A graph-based approach to API usage adaptation. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 302–321. ACM, New York (2010)
15. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J., Nguyen, T.N.: Recurring bug fixes in object-oriented programs. In: ICSE 2010: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 315–324. ACM, New York (2010)
16. Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: ESEC/FSE 2009: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 383–392. ACM, New York (2009)

17. Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Proceedings of the 11th European Software Engineering Conference held jointly with 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 35–44 (September 2007)
18. Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pp. 231–240. IEEE Computer Society, Washington, DC, USA (2006)
19. Williams, C.C., Hollingsworth, J.K.: Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.* 31(6), 466–480 (2005)
20. Xing, Z., Stroulia, E.: Refactoring detection based on umldiff change-facts queries. In: Proceedings of the 13th Working Conference on Reverse Engineering, pp. 263–274. IEEE Computer Society, Washington, DC, USA (2006)

Improving the Tokenisation of Identifier Names

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp

Computing Department and Centre for Research in Computing,
The Open University, Milton Keynes, United Kingdom

Abstract. Identifier names are the main vehicle for semantic information during program comprehension. Identifier names are tokenised into their semantic constituents by tools supporting program comprehension tasks, including concept location and requirements traceability. We present an approach to the automated tokenisation of identifier names that improves on existing techniques in two ways. First, it improves tokenisation accuracy for identifier names of a single case and those containing digits. Second, performance gains over existing techniques are achieved using smaller oracles. Accuracy was evaluated by comparing the output of our algorithm to manual tokenisations of 28,000 identifier names drawn from 60 open source Java projects totalling 16.5 MSLOC. We also undertook a study of the typographical features of identifier names (single case, use of digits, *etc.*) per object-oriented construct (class names, method names, *etc.*), thus providing an insight into naming conventions in industrial-scale object-oriented code. Our tokenisation tool and datasets are publicly available¹.

1 Introduction

Identifier names are strings of characters, often composed of one or more words, abbreviations and acronyms that describe actions and entities in source code. Identifier names are tokenised into their component words to support a wide range of activities in software development, maintenance and research, including concept location [16,14], to extract semantically useful information for other processes such as traceability [2], and the extraction of domain-specific ontologies [17], or to support investigations of the composition of identifier names [9,10].

Identifier naming conventions describe how developers *should* construct identifier names. The conventions typically provide mechanisms for identifying boundaries between component words either with separator characters, e.g. `get_text` (Eclipse), or internal capitalisation where the initial letter of the second and successive component words is capitalised, colloquially known as ‘camel case’, e.g. `getText` (OpenProj). The use of separator characters and internal capitalisation mean identifier names can be readily tokenised. However, a non-negligible proportion of identifier names (we found approximately 15%) are more difficult to tokenise accurately and reliably because they contain features such as upper

¹ <http://oro.open.ac.uk/28352/>

case acronyms, unconventional uses of capitalisation and digits, or are composed of characters of a single case. Upper case acronyms and words are delimited inconsistently, e.g. `setOSTypes` (jEdit) contains the acronym `OS`, `hasSVUID` (Google Web Toolkit) contains two acronyms, `SVU` and `ID`, concatenated, while `DAYSforMONTH` [7] relies on a change of case to mark a word boundary. Digits are found in some acronyms, e.g. `J2se` and `POP3`, and are also found as discrete tokens, thus there is no simple means of recognising a word boundary where a digit appears in an identifier name. Single case identifier names contain no readily identifiable word boundaries and in some instances, e.g. `ALTORENDSTATE` (JDK), have more than one plausible tokenisation based on dictionary words, which needs to be resolved. Further difficulties arise from the use of mixed case acronyms like `OSGi` and `DnD`, where the acronym is difficult to recover as a single token when used in the mixed case form, e.g. as in `isOSGiCompatible` (Eclipse), which lack conventional word boundaries.

Current approaches to identifier name tokenisation [7,8,15] report accuracies of around 96% for the tokenisation of unique identifier names. However, some approaches ignore identifier names containing digits [8,15], or treat digits as discrete tokens [7]. In this paper, we present a step-wise strategy to tokenising identifier names that improves on existing methods [7,8] in three ways. Firstly, we introduce a method for tokenising single case identifier names that addresses the problem of resolving ambiguous tokenisations and does not rely on the assumption that identifier names begin and end with known words; secondly, we implement and evaluate a method of tokenising identifier names containing digits that relies on an oracle and heuristics; and thirdly, we use an oracle created from published word lists [4] with 117,000 entries, which makes the solution easier to create and deploy than that described in [7] where the oracle consists of 630,000 entries harvested from 9,000 Java projects.

Improvements in identifier name tokenisation can have a big impact on the coverage of concept location and program comprehension tools because tokenisation accuracy is reported in terms of unique identifier names. Hence, even a 1% improvement of accuracy can have a radical effect (e.g. in concept location) if it affects those identifiers with many instances throughout the source code, which would otherwise lead to incorrect or missing concept locations. More importantly, by improving techniques for tokenising identifier names composed of characters of a single case and those containing digits, the coverage of concept location tools can be extended to include identifier names have previously been ignored or underused.

Identifier name tokenisation can also be used in IDE tools to support identifier name quality assurance. For example, some projects use tools like Checkstyle² to check conformance to programming conventions when source code is committed to the repository. Such tools typically only ensure typographical conventions, like the usage of word separators in names of constants, not lexical ones, like the usage of dictionary words and recognised abbreviations. Using tokenisation

² <http://checkstyle.sourceforge.net/>

to check whether an identifier name can be properly parsed would allow a more pro-active approach to ensuring the readability of source code.

The remainder of the paper is structured as follows. Section 2 consists of an exposition of the problems encountered when tokenising identifier names. In Section 3 we give an account of related work including the approaches taken by other researchers, before describing our approach to the problem in Section 4. In Section 5 we describe the experiments undertaken to evaluate our solution and compare it with existing solutions. In Sections 6 and 7 we discuss the results of our experiments and draw our conclusions.

2 The Identifier Name Tokenisation Problem

In this section we describe the practical problems encountered when trying to tokenise identifier names.

2.1 The Composition of Identifier Names

Programming languages and programming conventions constrain the content and form of identifier names. Programming languages impose hard constraints, most commonly that identifier names must consist of a single string³, where the initial character is not a digit, and are composed of a restricted set of characters. For the majority of programming languages, the set of characters permitted in identifier names consists of upper and lower case letters, digits, and some additional characters used as separators. An additional hard constraint imposed by languages such as Perl and PHP is that identifier names begin with specific non-alphanumeric characters used as sigils – signs or symbols – to identify the type represented by the identifier. For example, in Perl ‘\$’ denotes a scalar and ‘@’ a vector.

Programming conventions provide soft constraints in the form of rules on the parts of speech to be used in identifier names, how word boundaries should be constructed and often include the vague injunction that identifier names should be ‘meaningful’. Programming conventions typically advise developers to create identifier names with some means of identifying boundaries between words. Java, for example, employs two conventions [19]: constants are composed of words and abbreviations in upper case characters and digits separated by underscores (e.g. `FOO_BAR`), and may be described by the regular expression $U[DU]^*(S[DU]^+)^*$, where D represents a digit, S a separator character and U an upper case letter; and all other identifier names rely on internal capitalisation to separate component words (e.g. `fooBar`).

2.2 Tokenising Identifier Names

Programming conventions, though applied widely, are soft constraints and, consequently, are not applied universally. Thus, tools that tokenise identifier names

³ Smalltalk method names are a rare exception where the identifier name is separated to accommodate the arguments, e.g. `multiply: x by: y`.

need to provide strategies for splitting both conventionally and unconventionally constructed identifier names. Identifier names contain features such as separator characters, changes in case, and digits that have an impact on tokenisation. We discuss each feature before looking at the difficulties encountered when attempting to tokenise identifier names without separator characters or changes in case to indicate word boundaries.

Separator Characters. Separator characters – for example, the hyphen in Lisp and the full-stop, or period, in `R4` – can be used to separate the component words in identifier names. Accordingly, the identification of conventional internal boundaries in identifier names is straightforward, and the vocabulary used by the creator of the identifier name can be recovered accurately.

Internal Capitalisation. *Internal capitalisation*, often referred to as ‘camel case’, is an alternative convention for marking word boundaries in identifier names. The start of the second and subsequent words in an identifier name are marked with an upper case letter as in the identifier name `StyledEditorKit` (Java Library), where the boundary between the component words of an identifier name occurs at the transition between a lower case and an upper case letter, i.e. internally capitalised identifier names are of the form $U^?L^+(UL^+)^*$, where L represents a lower case letter, and the word boundary is characterised by the regular expression LU . The word boundary is easily detected and identifier names constructed using internal capitalisation are readily tokenised.

A second type of internal capitalisation boundary is found in practice. Some identifier names contain a sequence consisting of two or more upper case letters followed by at least one lower case letter, i.e. the sequence U^+UL^+ . We refer to this type of boundary as the *UCLC boundary*, where UCLC is an abbreviation of upper case to lower case. Most commonly, identifier names with a UCLC boundary contain capitalised acronyms, for example the Java library class name `HTMLEditorKit`. In these cases the word boundary occurs after the penultimate upper case letter of the sequence. However, identifier names have also been found [\[7\]](#) with the same characteristic sequence where the word boundary is marked by the change of case from upper case to lower case, for example `PBInitialize` (Apache Derby). Thus, identification of the UCLC boundary alone is insufficient to support accurate tokenisation [\[7\]](#).

Some identifier names mix the internal capitalisation and separator character conventions, e.g. `ATTRIBUTE_fontSize` (JasperReports). Despite being unconventional, such identifier names pose no further problems for tokenisation than those already given.

Digits. Digits occur in identifier names as part of an acronym or as discrete tokens. Where a digit or digits are embedded in the component word, as in the abbreviation J2SE, then the boundaries between tokens are defined by the internal capitalisation boundaries between the acronym and its neighbours. Abbreviations that have a bounding digit, e.g. POP3 and 3D, cannot be separated

⁴ <http://www.r-project.org/>

from other tokens where boundaries are defined by case transitions between alphabetical characters. Even if developers rigorously adopted the convention of only capitalising the initial character of acronyms advocated by Vermeulen [20], that would only help detect the boundary following a trailing digit (e.g. `Pop3Server`), it would not allow the assumption that a leading digit formed a boundary – that is it could not be assumed that UL^+DUL^+ may be tokenised as UL^+ and DUL^+ . In other words, because digits do not appear in consistent positions in acronyms, there is no simple rule that can be applied to tokenise identifier names containing acronyms that include digits. Similar complications arise where digits form a discrete component of identifier names, including the use of digits as suffixes (e.g. `index3`) and as homophone substitutions for prepositions (e.g. `html2xml`).

Single Case. Some identifier names are composed exclusively of either upper case (U^+) or lower case characters (L^+), or are composed of a single upper case letter followed by lower case letters (UL^+). Such identifier names are often formed from a single word. However, some, such as `maxprefwidth` (Vuze) and `ALTORENDSTATE` (JDK), are composed of more than one word. Lacking word boundary markers, multi-word single case identifier names cannot be tokenised without the application of heuristics or the use of oracles. A variant of the single case pattern is also found within individual tokens in identifier names like `notAValueOutputStream` (Java library), where the developer has created a compound, or failed to mark word boundaries. Accordingly some tokens require inspection and, possibly, further tokenisation. When tokenising identifiers composed of a single case there are two dangers: *ambiguity* and *oversplitting*.

Ambiguity. Some single case identifier names have more than one possible tokenisation. For example, `ALTORENDSTATE` is, probably, intended to be interpreted as `{ALT, OR, END, STATE}`. However, it may also be tokenised as `{ALTO, RENDS, TATE}` by a greedy algorithm that recursively searches for the longest dictionary word match from the beginning of the string, leaving the proper noun ‘Tate’ as the remaining token. A function of tokenisation tools is therefore to disambiguate multiple tokenisations.

Oversplitting. The term *oversplitting* describes the excessive division of tokens by identifier name tokenisation software [7], e.g. tokenising the single case identifier name `outputfilename` as `{out, put, file, name}`. The consequence of this form of oversplitting is that search tools for concept location would not identify that ‘output’ was a component of `outputfilename` without additional effort to reconstruct words from tokens.

Oversplitting is also practised by developers in two forms: one conventional, the other unconventional. Oversplitting occurs in conventional practice in class identifier names that are part of an inheritance hierarchy. Class identifier names can be composed of part or all of the super class identifier name that may consist of a number of tokens and an adjectival phrase indicating the specialisation. For example, the class identifier name `HTMLEditorKit` is composed

of part of the type name of its super class `StyledEditorKit` and the adjectival abbreviation `HTML`, yet would be tokenised as `{HTML, Editor, Kit}`. In this case the compound of the super type is potentially lost, but can be recovered by program comprehension tools. Developers also oversplit components of identifier names unconventionally by inserting additional word boundaries, which increases the difficulty of recovering tokens that reflect the developer's intended meaning. Common instances include the oversplitting of tokens containing digits such as `Http_1.1`, the demarcation of some common prefixes as separate words as in `SubString`, and the division of some compounds such as *metadata* and *uppercase*. In each case, a recognisable semantic unit is subdivided into components and the composite meaning is lost, and must be recovered by program comprehension tools [14].

In the following section we examine the literature on identifier name tokenisation and the approaches adopted by different researchers to solving the problems outlined above.

3 Related Work

Though the tokenisation of identifier names is a relatively common activity undertaken by software engineering researchers [1,2,3,6,9,11,14,16,18], few researchers evaluate and report their methodologies.

Feild *et al.* [8] conducted an investigation of the tokenisation of single case identifier names, or *hard words* in their terminology. Their experimental effort focused on splitting single case identifier names into component, or *soft*, words. For example, the hard word `hashtable` is constructed from the two soft words `hash` and `table`.

Feild *et al.* compared three approaches to tokenising identifier names – a random algorithm, a greedy algorithm and a neural network. The greedy algorithm applied a recursive algorithm to match substrings of identifier names to words found in the `ispell`⁵ dictionaries to identify potential soft words. For hard words that are composed of more than one soft word, the algorithm starts at the beginning and end of the string looking for the longest known word and repeats the process recursively for the remainder of the string. For example `outputfilename` is tokenised as `{output, filename}` from the beginning of the string and as `{outputfile, name}` from the end of the string on the first pass. The process is then repeated and the forward and backward components of the algorithm produce the same list of soft words, and thus the single tokenisation `{output, file, name}`. Where lists of soft words are different, the list containing the higher proportion of known soft words is selected.

Of the three approaches, the greedy algorithm was found to be the more consistent, tokenising identifier names with an accuracy of 75-81%. The greedy algorithm, however, was prone to oversplitting. The neural network was found to be more accurate, but only under particular conditions, for example when the training set of tokenisations was created by an individual.

⁵ <http://www.gnu.org/software/ispell/ispell.html>

In a related study Lawrie *et al.* [12] turned to expanding abbreviations to support identifier name tokenisation, and posed the question: how should an ambiguous identifier name such as `thenewestone` be divided into component soft words? Depending on the algorithm used there are a number of plausible tokenisations and no obvious way of selecting the correct one, e.g. `{the, newest, one}`, `{then, ewe, stone}`, and `{then, ewes, tone}`. Lawrie *et al.* suggested that the solution lies in a heuristic that relies on the likelihood of the soft words being found in the vocabulary used in the program's identifier names.

Enslin *et al.* expanded on these ideas in a tool named *Samurai* [7]. Samurai applies a four step algorithm to the tokenisation of identifier names.

1. Identifier names are first tokenised using boundaries marked by separator characters or the transitions between letters and digits.
2. The tokens from step 1 are investigated for the presence of changes from lower case to upper case (the primary internal capitalisation boundary) and split on those boundaries.
3. Tokens found to contain the UCLC boundary – as found in `HTMLEditor` – are investigated using an oracle to determine whether splitting the token following the penultimate upper case letter, or at the change from upper to lower case results in a better tokenisation.
4. Each token is investigated using a recursive algorithm with the support of an oracle to determine whether it can be divided further.

The oracle used in steps 3 and 4 was constructed by recording the frequency of tokens resulting from naive tokenisation based on steps 1 and 2 found in identifier names extracted from 9,000 Sourceforge projects. The oracle returns a score for a token based on its global frequency among all the code analysed and its frequency in the program being analysed. The algorithms in steps 3 and 4 are conservative. In step 3 the algorithm is biased to split the string following the penultimate upper case letter, and will only split on the boundary between upper and lower case where there is overwhelming evidence that the tokenisation is more frequent. The recursive algorithm applied in step 4 will only divide a single case string where there is strong evidence to do so, and also relies on lists of prefixes and suffixes⁶ to prevent oversplitting. For example, the token `listen` could be tokenised as `{list, en}` for projects where 'list' occurs as a token with much greater frequency than 'listen'. Samurai avoids such oversplitting by ignoring possible tokenisations where one of the candidate tokens, such as 'en', is found in the lists of prefixes and suffixes.

Enslin *et al.* also reproduced the 'greedy algorithm' reported by Feild *et al.* and compared the relative accuracies of the two techniques. The experiment used a reference set of 8,000 identifier names that had been tokenised by hand. The Samurai algorithm performed better than their implementation of the greedy algorithm, with an accuracy of 97%. The Samurai algorithm has some limitations which we discuss in the next section.

Madani *et al.* [15] developed an algorithm, derived from speech recognition techniques, to split identifier names that does not rely on conventional internal

⁶ Available from <http://www.cis.udel.edu/~enslin/samurai>

capitalisation boundaries. The approach tries to match substrings of an identifier name with entries in an oracle, both as a straightforward match and through a process of abbreviation expansion analogous to that used by a spell-checking program. Thus `idxcnt` would be tokenised as `{index, count}`. Furthermore, because the algorithm ignores internal capitalisation it can consistently tokenise component words such as `MetaData` and `metadata`. Madani *et al.* achieved accuracy rates of between 93% and 96% in their evaluations, which was better than naive camel case splitting in both projects investigated.

In the next section we describe our approach and how it differs from the above techniques.

4 Approach

The approaches described were found to tokenise 96-97% of identifier names accurately. However, there are limitations to each solution and issues with their implementation that make their application in practical tools difficult. Of the three approaches discussed, only Enslin *et al.* attempt to process identifier names containing digits. However, digits are isolated as separate tokens at an early stage of the Samurai algorithm so that meaningful acronyms such as `http11` are tokenised as `{http, 11}`. Samurai is also hampered by the amount of data collection required to create its supporting oracle.

We have implemented a solution to the problem of identifier name tokenisation that addresses the issues identified in current tools. The solution named INTT, or *Identifier Name Tokeniser Tool*, is part of a larger source code mining tool [5]. In particular, we have tried to ensure that the solution is relatively easy to implement and deploy, and is able to tokenise all types of identifier name. INTT applies naive tokenisation to identifier names that contain conventional separator character and internal capitalisation word boundaries. Tokens containing the UCLC boundary or digits are processed using heuristics to determine a likely tokenisation, and identifier names composed of letters of a single case are tokenised using an adaptation of the greedy algorithm described above.

The core tokenisation functionality of INTT is implemented in a JAR file so that it can be readily incorporated into other tools. The simple API allows the caller to invoke the tokeniser on a single string, and returns the tokens as an array. Thus front ends can range in sophistication from basic command line utilities that process individual identifier names to parser based tools that process source code. To support programming language independence the set of separator characters can be configured using the API, but the caller is responsible for removing any sigils from the identifier name. However, INTT has only been tested on identifier names extracted from Java source code.

In summary, our algorithm consists of the following steps, which we discuss in detail below:

1. Identifier names are tokenised using separator characters and the internal capitalisation boundaries.
2. Any token containing the UCLC boundary is tokenised with the support of an oracle.

3. Any identifier names with tokens containing digits are reviewed and tokenised using an oracle and a set of heuristics.
4. Any identifier name composed of a single token is investigated to determine whether it is a recognised word or a neologism constructed from the simple addition of known prefixes and suffixes to a recognised word.
5. Any remaining single token identifier names are tokenised by recursive algorithms. Candidate tokenisations are investigated to reduce oversplitting, before being scored with weight being given to tokens found in the project-specific vocabulary.

4.1 Oracles

To support the tokenisation of identifier names containing the UCLC boundary, digits and single case identifier names, we constructed three oracles: a list of dictionary words, a list of abbreviations and acronyms, and a list of acronyms containing digits. The list of dictionary words consists of some 117,000 words, including inflections and American and Canadian English spelling variations, from the SCOWL package word lists up to size 70, the largest lists consisting of words commonly found in published dictionaries [4]. We added a further 120 common computing and Java terms, e.g. ‘arity’, ‘hostname’, ‘symlink’, and ‘throwable’. Previous work [5] included analysis of which identifier names did not correspond to dictionary words and found that several known computing terms were unrecognised. The list of computing terms was hence constructed iteratively over the analysed projects, using the criterion that any word added should be a known, non-trivial computing term. Each oracle was implemented using a Java HashSet so that lookups are performed in constant time.

The use of dictionaries imposes a limitation on the accuracy of the resulting tokenisation because a natural language dictionary cannot be complete. We addressed this limitation by adopting a method to incorporate the lexicon of the program being processed in an additional oracle, which takes a step towards resolving the issue highlighted in Lawrie *et al.*’s question of how to resolve ambiguous tokenisations for identifier names such as `thenewestone` [12]. Tokens resulting from the tokenisation of conventionally constructed identifier names are recorded in a temporary oracle to provide a local – i.e. domain- or project-specific – vocabulary that is employed to support the tokenisation of single case identifier names. For example, tokens extracted from identifier names such as `pageIdx` and `lineCnt` can be used to support the tokenisation of an identifier name like `idxcnt` as `{idx, cnt}`.

INTT is also able to incorporate alternative lists of dictionary words in its oracle, and is, thus, potentially language independent. INTT relies on Java’s string and character representations, which default to the UTF-16 unicode character encoding standard. So, INTT is able to support dictionaries, and thus tokenise identifier names created using natural languages where all the characters, including accented characters, can be represented using UTF-16 (subject to the constraints on identifier name character sets imposed by the programming language). However, as INTT was designed with the English language and English morphology in mind, adaptation to other languages may not be straightforward.

4.2 Tokenising Conventionally Constructed Identifier Names

The first stage of INTT tokenises identifier names using boundaries marked by separator characters and on the conventional lower case to upper case internal capitalisation boundaries. Where the UCLC boundary is identified, INTT investigates the two possible tokenisations: the conventional internal capitalisation where the boundary lies between the final two letters of the upper case sequence, e.g. as found in `HTMLEditorKit`; and the boundary following the sequence of upper case letters, as in `PBinitialize`. The preferred tokenisation is that containing more words found in the oracle. Where this is not a discriminant, tokenisation at the internal capitalisation boundary is preferred.

Following the initial tokenisation process, identifier names are screened to identify those that require more detailed processing. Identifier names found to contain one or more tokens with digits are tokenised using heuristics and an oracle. Identifier names composed of letters of a single case are tokenised, if necessary, using a variant of the greedy algorithm [12]. These processes are described in detail below.

4.3 Tokenising Identifier Names Containing Digits

In Section 2 we outlined the issues concerning the tokenisation of identifier names containing digits. We identified three uses of digits in identifier names: in acronyms (e.g. `getX500Principal` (JDK)), as suffixes (e.g. `typeList2` (JDK, Java libraries and Xerces)) and as homophone substitutes for prepositions (e.g. `ascii2binary` (JDK and Java libraries)). In the latter two cases the digit, or group of digits, forms a discrete token of the identifier, and if identified correctly the identifier name may be tokenised with relative ease. Acronyms containing digits are more problematic. We have identified two basic forms of acronym: those with an embedded digit, e.g. `J2SE`, and those with one or more bounding digits, e.g. `3D`, `POP3` and `2of7`.

Acronyms with embedded digits are bounded by letters and can be tokenised correctly by relying on internal capitalisation boundaries alone. For example, the method identifier name `createJ2SEPlatform` (Netbeans) can be tokenised as `{create, J2SE, Platform}` without any need to investigate the digit. Acronyms with leading or trailing digits cannot easily be tokenised, and neither can those with bounding digits. We made a special case of acronyms with bounding digits. While they could be tokenised on the assumption that the digits were discrete tokens, we decided that the very few instances of acronyms with bounding digits found in the subject source code were better seen as discrete tokens from a program comprehension perspective. Indeed all the instances we found were noun phrases describing mappings, `1to1`, or bar code encoding schemes `2of7`.

With the exception of the embedded digit form of acronym there is no general rule by which to tokenise identifier names containing digits. Accordingly we created an oracle from a list of common acronyms containing digits and developed a set of heuristics to support the tokenisation of identifier names containing digits.

Identifier names are first tokenised using separator characters and the rules for internal capitalisation. Where a token is found to contain one or more digits it is investigated to determine whether it contains an acronym found in the oracle. Where the acronym is recognised the identifier name is tokenised so that the acronym is a token. For example, `Pop3StoreGBean` can be tokenised using internal capitalisation as `{Pop3Store, G, Bean}`. The tokens are then investigated for known digit containing acronyms and tokenised on the assumption that `Pop3` is a token, resulting in the tokenisation of `{Pop3, Store}`.

Where known acronyms are not found, the digit containing token is split to isolate the digit and an attempt made to determine whether the digit is a suffix of the left hand textual fragment, a prefix of the right hand one, or a discrete token. We employ the following heuristics:

1. If the identifier name consists of a single token with a trailing digit, then the digit is a discrete token, e.g. `radius2` (Netbeans) is tokenised as `{radius, 2}`.
2. If both the left and right hand tokens are both words or known acronyms the digit is assumed to be a suffix of the left hand token, e.g. `eclipse21Profile` (Eclipse) is tokenised as `{eclipse21, Profile}`.
3. If both the left and right hand tokens are unrecognised the digit is assumed to be a suffix of the left hand token, e.g. `c2tnb431r1` (Geronimo and JDK) is tokenised as `{c2, tnb431, r1}`.
4. If the left hand token is a known word and the right hand token is unrecognised, then the digit is assumed to be a prefix of the right hand token, e.g. `is9x` (Geronimo) is tokenised as `{is, 9x}`.
5. If the digit is either a 2 or 4 and the left and right hand fragments are known words, the digit is assumed to be a homophone substitution for a preposition, and thus a discrete token, e.g. `ascii2binary` is tokenised as `{ascii, 2, binary}`. It is trivial for the application that calls our tokenisation method to expand the digit into 'to' or 'for', if deemed relevant for the application.

4.4 Tokenising Single Case Identifier Names

To tokenise single case identifier names we adapted the greedy algorithm developed by Feild *et al.* [8]. We identified two areas of the greedy algorithm that required modification to suit our purposes. Firstly, because the algorithm is greedy, it may fail to identify more accurate tokenisations in particular circumstances. For example, the algorithm finds the longest known word from beginning and end of the string, so `thenewestone` would be tokenised as `{then, ewes, tone}` by the forward pass, and as `{thenewe, stone}` by the backward pass. Secondly, the algorithm assumes that the string to be processed begins or ends with a recognised soft word and therefore cannot locate soft words in a string that both begins and ends with unrecognised words.

Our adaptation of the greedy algorithm is implemented in two forms: greedy and greedier. The greedy algorithm assumes that the string being investigated either begins or ends with a known soft word and the greedier algorithm is only invoked when the greedy algorithm cannot tokenise the string.

Algorithm 1. INTT greedy algorithm: forward tokenisation pass

```

1: procedure GREEDYTOKENISEFORWARDS(s)
2:   candidates                                     ▷ a list of lists
3:   for i ← 0, length(s) do
4:     if s[0, i] is found in dictionary then
5:       rightCandidates ← greedyTokeniseForwards(s[i + 1, length(s)])
6:       for all lists of tokens in rightCandidates do
7:         add s[0, i] to beginning of list
8:         add list to candidates
9:       end for
10:    end if
11:  end for
12:  if candidates is empty then
13:    create new list with s as member
14:    add list to candidates
15:  end if
16:  return candidates
17: end procedure

```

Prior to the application of the greedy algorithm, strings are screened to ensure that they are not recognised words or simple neologisms. The check for simple neologisms uses lists of prefixes and suffixes to check that strings are not composed of a combination of, for example, a known prefix followed by a known word. This allows identifier names such as `discontiguous` (Java Libraries, JDK and NetBeans) to be recognised as words, despite them not being recorded in the dictionary. The greedy algorithm iterates over the characters of the identifier name string forwards (see Algorithm 1) and backwards. On each iteration, the substring from the end of the string to the current character is tested using the dictionary words and acronyms oracles to establish whether the substring is a known word or acronym. When a match is found the soft word is stored in a list of candidates and the search invoked recursively on the remainder of the string. Where no word can be identified the remainder of the string is added to the list of candidates.

When the greedy algorithm is unable to tokenise the string, the greedier algorithm is invoked. The greedier algorithm attempts to tokenise a string by creating a prefix of increasing length from the initial characters and invokes the greedy algorithm on the remainder of the string to identify known words (see Algorithm 2). For example, for the string `cdoutputef`, `c` is added to a list of candidates and the greedy algorithm invoked on `doutputef`, then the prefix `cd` is tried and the greedy algorithm invoked on `outputef` resulting in the tokenisation `{cd, output, ef}`. This process is repeated, processing the string both forwards and backwards until the prefix and suffix are one character less than half the length of the string being tokenised, which allows the forward

and backward passes to find small words sandwiched between long prefixes and suffixes, while avoiding redundant processing. For example in the string `yyytozz` both the forwards and backwards passes will recognise `to`, and in the string `yyytozz` the backwards pass will recognise `to`.

Algorithm 2. INTT greedier algorithm: backwards tokenisation pass

```

1: procedure GREEDIERTOKENISEBACKWARDS(s)
2:   candidates                                     ▷ a list of lists
3:   for i ← length(s), length(s)/2 do
4:     leftCandidates ← greedyTokeniseBackwards(s[0, i − 1])
5:     for all lists of tokens in leftCandidates do
6:       add s[i, length(s)] to beginning of list
7:       add list to candidates
8:     end for
9:   end for
10:  return candidates
11: end procedure

```

Each list of candidate component words is scored according to the percentage of the component words found in the dictionaries of words and abbreviations, and the program vocabulary – i.e. component words found in identifier names in the program that were split using conventional internal capitalisation boundaries and separator characters. The percentage of known words is recorded as an integer and a weight of one added for each word found in the program vocabulary. For example, suppose splitting `thenewestone` resulted in two candidate sets `{the, newest, one}` and `{then, ewe, stone}`. All the words in both sets are found in the dictionaries used and thus each set of candidates score 100. However, suppose `newest` and `one` are found in the list of identifier names used in the program, so two is added to the score of the first set, and that is selected as the preferred tokenisation.

The algorithm, because of its intensive search for candidate component words, is prone to evaluating an oversplit tokenisation as a better option than a more plausible tokenisation. To reduce oversplitting, each candidate tokenisation is examined prior to scoring to determine whether adjacent soft words can be concatenated to form dictionary words. Where this is the case the oversplit set of tokens is replaced by the concatenated version. For example `outputfile` would be tokenised as `{output, file}` and `{out, put, file}`. Following the check for oversplitting, the first two tokens of the latter tokenisation would be concatenated making the two tokenisations identical, allowing one to be discarded.

The key advantage offered by the greedy and greedier algorithms are that a single case identifier name can be tokenised without the requirement that it begins or ends with a known word. For example, Feild *et al.*'s greedy algorithm cannot tokenise identifier names like `lboundsb` unless 'b' or 'l' are separate entries in the oracle. Samurai can only tokenise `lboundsb` if 'l' or 'lbound' are found

as separate tokens in the oracle. Our algorithm can tokenise `lbounds` using a dictionary where ‘bounds’ is an entry.

In the following section we evaluate the accuracy of our identifier name tokenisation algorithm and compare its performance with Samurai and Feild *et al.*’s greedy algorithm.

5 Experiments and Results

To evaluate our approach and compare its performance with existing tools we adopted a similar procedure to that used by Feild *et al.* [8] and Enslin *et al.* [7]. However, instead of using a single test set of identifier names, we created seven test sets consisting of 4,000 identifier names each, extracted at random from a database of 827,475 unique identifier names from 16.5MSLOC⁷ of Java from 60 projects, including ArgoUML, Cobertura, Eclipse, FindBugs, the Java libraries and JDK, Kawa and Xerces⁸. One test set consists of identifier names selected at random from the database. Five test sets consist of random selections of particular *species* of identifier name – we use the term *species* to identify the role the identifier name plays in the programming language, such as a class or method name. The seventh set consists of identifier names composed of a single case only (see Table II).

Table 1. Distribution of identifier name categories in datasets

Dataset Description		Conventional	Digits	Single Case	UCLC
A	Random identifier names	2414	467	1011	106
B	Class names	3133	185	113	569
C	Method names	3459	116	184	151
D	Field names	2717	401	818	64
E	Formal arguments	2754	250	961	34
F	Local variable names	2596	349	1021	34
G	Single case	0	0	4000	0

Each test set of 4,000 identifier names was tokenised manually by the first author to provide reference sets of tokenisations. The resulting text files consist of lines composed of the identifier name followed by a tab character and the tokenised form of the identifier name, normalised in lower case, with each token

⁷ Obtained using Sloccount <http://www.dwheeler.com/sloccount/>

⁸ A complete list of the projects analysed is available with the INTT library at <http://oro.open.ac.uk/28352/>

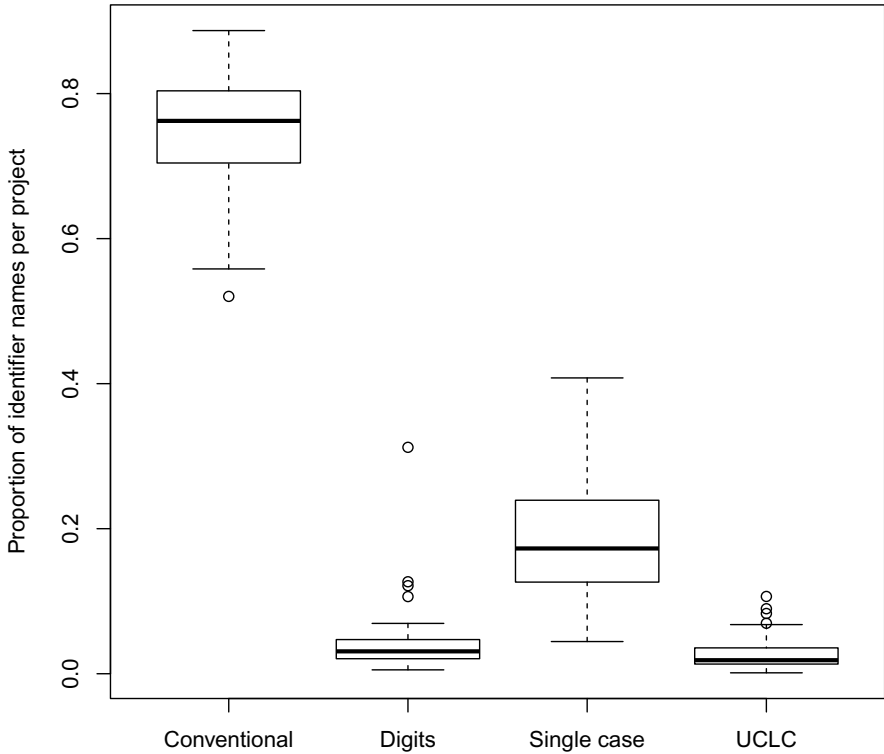


Fig. 1. Distribution of the percentage of unique identifier names found in each category for sixty Java projects

separated by a dash, e.g. `HTMLEditorKit(tab)html-editor-kit`. Bias may have been introduced to our experiment by the reference tokenisations having not been created independently and we discuss the implications below in Subsection [5.4](#) Threats to Validity.

The identifier names in the test sets were classified using four largely mutually exclusive categories that reflect particular features of identifier name composition related to the difficulty of accurate tokenisation. The categories are:

- **Conventional** identifier names are composed of groups of letters divided by internal capitalisation (lower case to upper case boundary) or separator characters.
- **Digits** identifier names contain one or more digits.
- **Single case** identifier names are composed only of letters of the same case, or begin with a single upper case letter with the remaining characters all lower case.
- **UCLC** identifier names contain two or more contiguous upper case characters followed by a lower case character.

Identifier names are categorised by first testing for the presence of one or more digits, then testing for the UCLC boundary. Consequently the digits category may contain some identifier names that also have the UCLC boundary. In the seven test sets there are a total of 1768 identifier names containing digits, of which 62 also contain a UCLC boundary. The classification system is intended to allow the exclusion of identifier names containing digits from evaluations of those tools that do not attempt realistic tokenisation of such identifier names, and to allow evaluation of our approach to tokenising identifier names containing digits. The distribution of the four categories of identifier names in each of the datasets is given in Table 1.

We also surveyed the 60 projects in our database. Figure 1 shows the distribution of each category as a proportion of the total number of unique identifier names in each application. Identifier names containing only conventional boundaries are by far the most common form of identifier name found in all the projects surveyed. A significant proportion of single case identifier names are found in most projects, and around 10% of identifier names contain digits or the UCLC boundary. Table 2 gives a breakdown of the proportion of unique identifier names in each category across all 60 projects for each species of identifier. Test sets B to F reflect the most common species, with the exception of constructor names which are lexically identical to class identifier names, but differ in distribution because not all classes have an explicitly declared constructor, while others have more than one.

Table 2. Percentage distribution of identifier name categories by species

Species	Conventional	Digits	Single case	UCLC	Overall %
Annotation	70.4	0.2	25.6	3.8	0.1
Annotation member	49.8	0.5	49.5	0.2	<0.1
Class	79.8	4.1	2.9	13.2	9.8
Constructor	79.8	3.5	3.1	13.5	7.2
Enum	73.4	0.5	19.4	6.7	0.1
Enum constant	55.9	10.2	33.6	0.2	0.8
Field	86.1	6.0	6.2	1.7	27.1
Formal argument	81.8	3.0	14.2	0.1	8.1
Interface	59.3	2.6	6.4	31.7	1.5
Label name	59.1	15.7	25.0	0.1	0.1
Local variable	82.4	3.8	12.6	1.2	16.9
Method	91.6	2.9	1.6	3.9	28.4
Total	84.9	4.1	6.4	4.6	

Table 2 shows that identifier names containing digits and those containing UCLC boundaries constitute nearly 9% of all the identifier names surveyed. Class, constructor and interface identifier names, the most important names for high level global program comprehension, have a relatively high incidence of identifier names containing the UCLC boundary – 13% for class and constructor identifier names and 32% for interface identifier names. In other words, approximately 20% of class names and 40% of interface names require more sophisticated heuristics to determine how to tokenise them.

We evaluated the performance of INTT by assessing the accuracy with which the test sets of identifier names were tokenised, and by comparing INTT with an implementation of the Samurai algorithm, both in terms of accuracy and the relative strengths and weaknesses of the two approaches.

5.1 INTT

We used INTT to tokenise the identifier names in each of the seven datasets. The accuracy of the tokenisations was automatically checked against the reference tokenisations for each dataset using a small Java program. A percentage accuracy score calculated for INTT’s overall performance and for each species of identifier name. A percentage accuracy was also calculated for each of the four structural categories found in each set of identifier names, see Table 3. (The results for dataset G are reported in Subsection 5.3)

Table 3. Percentage accuracies for INTT

Dataset		Conventional	Digits	Single case	UCLC	Overall	Without digits
A	Random identifier names	97.3	95.9	97.4	85.8	96.9	97.0
B	Class names	98.3	85.4	92.4	92.1	96.5	97.1
C	Method names	97.1	63.8	96.8	92.7	96.0	96.9
D	Field names	97.5	88.7	96.4	87.5	96.3	97.1
E	Formal arguments	98.8	94.4	93.4	79.4	97.0	97.2
F	Local variable names	98.2	94.3	92.0	85.3	96.2	96.3

INTT was found to have an overall accuracy of 96-97%, which improves marginally when identifier names containing digits are excluded. Identifier names containing digits are tokenised with an accuracy in excess of 85% for three of the six data sets A–F. However, accuracy drops to 64% for method identifier names containing digits. Inspection of the tokenisations for class and method names show that there are two contributing factors: firstly, the assumption that a

recognised acronym containing digits always takes precedence over the heuristics when determining a tokenisation led to incorrect tokenisations in some instances and, secondly, some oversplitting of textual tokens occurs. An example of the former is the method name `replaceXpp3DOM` (NetBeans) which was tokenised as `{replace, Xpp, 3D, OM}` on the basis that 3D is a known acronym containing digits. Applying the heuristics alone, however, would have found the correct tokenisation of `{replace, Xpp3, DOM}`.

The overall percentage accuracy for each dataset is comparable with the accuracies reported for the Samurai tool [7] (97%) and by Madani *et al.* [15] (93-96%). The breakdowns for each structural type of identifier name show that INTT performs less consistently for identifier names containing digits and for those containing the UCLC boundary.

5.2 Comparison with Samurai

To make a comparison with the work of Enslin *et al.* we developed an implementation of the Samurai tool based on the published pseudocode and textual descriptions of the algorithm [7]. The implementation processed the seven test sets of identifier names and the resulting tokenisations were scored for accuracy against the reference tokenisations. The results are shown in Table 4 with the exception of the single case dataset G, which is reported below in Subsection 5.3. The overall accuracy figure given for our implementation of the Samurai algorithm in Table 4 excludes identifier names with digits, and should be compared with the figures in the rightmost column of Table 3. Samurai’s treatment of digits as discrete tokens leads to an accuracy of 80% or more for all but class and method identifier names, where accuracy falls to 45% and 55% respectively.

Our implementation of the Samurai algorithm performs less well than the original [7]. On inspecting the tokenisations we found more oversplitting than we had anticipated. There are a number of factors that could contribute to the

Table 4. Percentage accuracies for Samurai

Dataset		Conventional	Digits	Single case	UCLC	Without digits
A	Random identifier names	93.3	92.9	69.1	82.1	86.3
B	Class names	94.0	44.9	86.3	81.5	91.7
C	Method names	92.8	55.2	88.8	83.4	92.3
D	Field names	91.3	78.8	78.2	73.4	87.7
E	Formal arguments	94.8	88.4	75.0	64.7	89.4
F	Local variable names	92.7	86.2	67.7	70.6	85.4

observed difference in performance, which we discuss in Subsection 5.4 Threats to Validity.

5.3 Single Case Identifier Names

Both INTT and Samurai contain algorithms for tokenising single case identifier names that are intended to improve on Feild *et al.*'s greedy algorithm. To compare the two tools we extracted a data set of 4,000 random single case identifier names from our database. All the identifier names consist of a minimum of eight characters: 2,497 are composed of more than one word or abbreviation, the remainder are either single words found in the dictionary or have no obvious tokenisation.

We implemented the greedy algorithm developed by Feild *et al.* following their published description [8], to provide a baseline of performance from which we could evaluate the improvement in performance represented by INTT and Samurai. The supporting dictionary for the Feild *et al.*'s greedy algorithm was constructed from the English word lists provided with ispell v3.1.20, the same version used by Feild *et al.*. We replaced their stop-list and list of abbreviations, with the same list of abbreviations used in INTT and the additional list of terms that are included in INTT's dictionary.

Enslin *et al.* found that Samurai and greedy both had their strengths. Samurai is a conservative algorithm that tokenises identifier names only when the tokenisation is a very much better option than not tokenising. As a result, the greedy algorithm correctly tokenised identifier names that Samurai left intact. However, the greedy algorithm was more prone to oversplitting than the more conservative Samurai [7].

The 4,000 single case identifier names were tokenised with 78.4% accuracy by our implementation of the 'greedy' algorithm, with 70.4% accuracy by our implementation of Samurai, and with 81.6% accuracy by INTT.

5.4 Threats to Validity

The threats to validity in this study are concerned with construct validity and external validity. We do not consider internal validity because we make no claims of causality. Similarly, we do not consider statistical conclusion validity, because we have not used any statistical tests.

Construct Validity. There are two key concerns regarding construct validity: the possibility of bias being introduced through manual tokenisation of identifier names used to create sets of reference tokenisations; and the observed difference in performance between our implementation of Samurai and the accuracy reported for the original implementation [7].

That we split the identifier names for the reference tokenisations ourselves may have introduced a bias towards tokenisations that favour our tool. We guarded against this during the manual tokenisation process as much as possible, and conducted a review of the reference sets to look for any possible bias and revised

any such tokenisations found. Of the related works [8,7,15] only Enslin *et al.* used a reference set of tokenisations created independently.

We have identified three factors that may explain the reduced accuracy achieved by our implementation of Samurai in comparison to the reported accuracy of the original. When implementing the Samurai algorithm, we took all reasonable steps, including extensive unit testing, to ensure our implementation conformed to the published pseudo code and text descriptions [7]. However, it is possible that we may have inadvertently introduced errors. There is the possibility that computational steps may have inadvertently been omitted from the published pseudo code description. The third possibility is that the scoring formula used in Samurai to identify preferable tokenisations, which was derived empirically, may not hold for oracles composed of fewer tokens with lower frequencies. The oracle used in our implementation of Samurai was constructed using identifier names found in 60 Java projects, much fewer than the 9,000 projects Enslin *et al.* used as the basis for their dictionary. Our version of the Samurai oracle contains 61,580 tokens, with a total frequency of 3 million. In comparison the original Samurai oracle was created using 630,000 tokens with a total frequency of 938 million.

External Validity. External validity is concerned with generalisations that may be drawn from the results. Our experiments were conducted using identifier names extracted from Java source code only. Although we cannot claim any accuracy values for other programming languages, we would expect results to be similar for programming languages with similar programming conventions, because our tokenisation approach is independent of the programming language. Our experiments were also conducted on identifier names constructed using the English language. While the techniques and the tool we developed can be applied readily to identifier names in other natural languages, some of the heuristics, in particular the treatment of ‘2’ and ‘4’ as homophone substitutions for prepositions, may need to be revised for non-English natural languages.

6 Discussion

One of our primary motivations for adopting the approach described above was a concern over the computing resources, both in terms of time and space that were being devoted to solving the problem of identifier name tokenisation. The approach taken by Madani *et al.* processes each identifier name in detail and is thus relatively computationally intensive, while the Samurai algorithm relies on harvesting identifier names from a large body of existing source code – a total of 9,000 projects – to create the supporting oracle. Like Samurai, we process identifier names selectively and reserve more detailed processing for those identifier names assumed to be more problematic. However, we achieve levels of accuracy similar to the published figures for Samurai using a smaller oracle constructed, largely, from readily available components such as the SCOWL word lists.

6.1 Identifier Names Containing Digits

We demonstrated an approach to tokenising identifier names containing digits that achieves an accuracy of 64% at worst and most commonly 85%-95%. The only tool available for comparison was our implementation of the Samurai algorithm, which takes a simple and unambiguous approach to tokenising identifier names containing digits and achieves, an accuracy that is consistently between 10% and 3% less than that achieved by INTT, with the exception of class identifier names where Samurai's treatment of digits as discrete tokens results in an accuracy of 45%, some 40% less than INTT.

While we are largely satisfied with having achieved such high rates of accuracy, there is room for improvement. Inspection of INTT's output showed that some inaccurate tokenisations could be attributed to incorrect tokenisation of textual portions of the identifier name. However, they also showed that some of our heuristics for identifying how to tokenise around digits require refinement. One possibility is the introduction of a specific heuristic for tokens of the form 'v5', signifying a version number, so that they are tokenised consistently. We found that though most were tokenised accurately, some identifier names, for example `SPARCV9FlushwInstruction` (JDK), were not. The difficulty appears not to be the digit alone, but that the digit in combination with the letter is key to accurate tokenisation. Other incorrect tokenisations occurred where identifier names such as `replaceXpp3DOM` contain a known acronym. The solution in such cases appears to be to choose between the tokenisation resulting from using recognised acronyms, and that arising from the application of the heuristics alone.

6.2 Limitations

No current approach tokenises all identifier names accurately. Indeed, accurate tokenisation of all identifier names may only be possible with some projects where a given set of identifier naming conventions are strictly followed. However, we would argue that there are a number of barriers to tokenisation that are difficult to overcome, and outside the control of those processing source code to extract information. An underlying assumption of the approaches taken to identifier name tokenisation is that identifier names contain semantic information in the form of words, abbreviations and acronyms and that these can be identified and recovered. Developers, however, do not always follow identifier naming conventions and building software that can process all the forms of identifier names that developers can dream up is most likely impossible and would require a great deal of additional effort for a minimal increase in accuracy. For example, `is0x8000000000000000L` (Xerces) is an extremely unusual form of identifier name – the form is seen only three times⁹ in the 60 projects we surveyed – which would require additional functionality to parse the hexadecimal number in order to tokenise the identifier name accurately.

⁹ NetBeans unit tests include the method names `test0x01` and `test0x16`.

Another limitation arises from neologisms and misspelt words. Neologisms found in the single case test set include ‘devoidify’, ‘detokenated’, ‘discontiguous’, ‘grandcestor’, ‘indentator’, ‘pathinate’ and ‘precisify’. With the exception of ‘grandcestor’ these are all formed by the unconventional use of prefixes and suffixes with recognised words or morphological stems. Some, e.g. ‘discontiguous’ are vulnerable to oversplitting by the greedy algorithm, and algorithms based on it. Others may cause problems when concatenated with other words in single case identifier names where a plausible tokenisation is found to span the intended boundary between words.

Samurai and INTT both guard against oversplitting neologisms by using lists of prefixes and suffixes. INTT identifies single case identifier names found to be formed by a recognised word in combination with either or both a known prefix or suffix and does not attempt to tokenise them. Samurai tries to tokenise all single case identifier names, but rejects possible tokenisations where one of the resulting tokens would be a known prefix or suffix. All of the neologisms listed would be recognised as single words by both approaches. However, INTT would not recognise ‘precisify’ as a neologism resulting from concatenation and would try to tokenise it.

Tools that use natural language dictionaries as oracles will try to tokenise a misspelt word, whether it is found in isolation or concatenated with another word, as a single case identifier name. The majority of observed misspellings result from insertion of an additional letter, omission of a letter or transposition of two letters. Precisely the sort of problem that can be readily identified by a spell checker. For example, `position` (NetBeans) is oversplit by both INTT and the greedy algorithm as `{pos, sit, ion}` and `{poss, it, ion}`, respectively. Samurai also oversplits `position` probably because of a combination of the relative rarity of the spelling mistake, the more common occurrence of the token `poss` (AspectJ, Eclipse, Netbeans, and Xalan). A step towards preventing some oversplitting of misspelt words could be achieved through the use of algorithms applied in spell-checking software, such as the Levenshtein distance [13].

Inspection of the tokenisations of the test sets for each tool show that the greedy algorithm is prone to oversplitting neologisms particularly where a suffix such as ‘able’ that is also a word has been added to a dictionary word, e.g. `zoomable` (JFreeChart). Greedy also cannot consistently tokenise identifier names that start and end with abbreviations not found in its dictionary, e.g. `tstampff` (BORG Calendar), and cannot differentiate between ambiguous tokenisations. Indeed, Feild *et al.* provide no description of how to differentiate between tokenisations that return identical scores [8]. In our implementation of the greedy algorithm, the tokenisation resulting from the backward pass is selected in such situations, because English language inflections, particularly the single ‘s’, can be included by the forward pass of the algorithm. For example, `debugstackmap` (JDK) is tokenised incorrectly as `{debugs, tack, map}` by the forward pass and correctly as `{debug, stack, map}` by the backward pass. The backward pass is also prone to incorrect tokenisations, though from inspection of the test set this is much less common. For example, the reverse pass tokenises

`commonkeys` (JDK) as `{com, monkeys}`, using `ispell` word lists where ‘com’ is listed as a word.

Tools such as `INTT` and `Samurai` work on the assumption that developers generally follow identifier naming conventions and that computational effort is required for exceptions that can be identified. As noted in our description of the problem (see Section 2) the assumption is an approximation. There are many cases where the conventions on word division are broken, or are used in ways that divide the elements of semantic units so as to render them meaningless. In other words, a key issue for tokenisation tools is that word divisions, be they separator characters or internal capitalisation, can be misleading and are thus not always reliable. Consequently, meaningful tokens may need to be reconstructed by concatenating adjacent tokens.

7 Conclusions

Identifier names are the main vehicle for semantic information during program comprehension. The majority of identifier names consist of two or more words or acronyms concatenated and therefore need to be tokenised to recover their semantic constituents, which can then be used for tool-supported program comprehension tasks, including concept location and requirements traceability. Tool-supported program comprehension is important for the maintenance of large object-oriented software projects where cross-cutting concerns mean that concepts are often not located in a single class, but are found diffused through the source code.

While identifier naming conventions should make the tokenisation of identifier names a straightforward task, they are not always clear, particularly with regard to digits, and developers do not always follow conventions rigorously, either using potentially ambiguous word division markers or none at all. Thus accurate identifier name tokenisation is a challenging task.

In particular, the tokenisation of identifier names of a single case is non-trivial and there are known limitations to existing methods, while identifier names containing digits have been largely ignored by published methods of identifier name tokenisation. However, these two forms of identifier name occur with a frequency of 9% in our survey of identifier names extracted from 16.5 MSLOC of Java source code, demonstrating the need to improve methods of tokenisation.

In this paper we make two contributions that improve on current identifier name tokenisation practice. First, we have introduced an original method for tokenising identifier names containing digits that can achieve accuracies in excess of 90% and is a consistent improvement over a naive tokenisation scheme. Second, we demonstrate an improvement on current methods for tokenising single case identifier names, on the one hand in terms of improved accuracy and scope by tokenising forms of identifier name that current tools cannot, and on the other hand in terms of resource usage by achieving similar or better accuracy using an oracle with less than 20% of the entries. Furthermore, the oracle we used can be constructed easily from available components, whereas the `Samurai` algorithm relies on identifier names harvested from 9,000 Java projects.

We make two further contributions. Firstly, INTT, written in Java, is available for download¹⁰ as a JAR file with an API that allows the identifier name tokenisation functionality described in this paper to be integrated into other tools. Secondly, the data used in this study is made available as plain text files. The data consists of the seven test datasets of 28,000 identifier names together with the manually obtained reference tokenisations, and 1.4 million records of over 800,000 unique identifier names in 60 open source Java projects, including information on the identifier species. By making these computational and data resources available, we hope to contribute to the further development of identifier name based techniques (not just tokenisation) that help improve software maintenance tasks.

Acknowledgements. We would like to thank the anonymous reviewers on the ECOOP 2011 Program Committee, and Tiago Alves and Eric Bouwers for their thoughtful comments that have helped improve this paper.

References

1. Abebe, S., Tonella, P.: Natural language parsing of program element names for concept extraction. In: 18th Int'l Conf. on Program Comprehension, pp. 156–159. IEEE, Los Alamitos (2010)
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering* 28(10), 970–983 (2002)
3. Antoniol, G., Gueheneuc, Y.G., Merlo, E., Tonella, P.: Mining the lexicon used by programmers during software [sic] evolution. In: Proc. of Int'l Conf. on Software Maintenance, pp. 14–23. IEEE, Los Alamitos (2007)
4. Atkinson, K.: SCOWL readme (2004), <http://wordlist.sourceforge.net/scowl-readme>
5. Butler, S., Wermelinger, M., Yu, Y., Sharp, H.: Exploring the influence of identifier names on code quality: an empirical study. In: Proc. of the 14th European Conf. on Software Maintenance and Reengineering, pp. 159–168. IEEE Computer Society, Los Alamitos (2010)
6. Caprile, B., Tonella, P.: Nomen est omen: analyzing the language of function identifiers. In: Proc. Sixth Working Conf. on Reverse Engineering, pp. 112–122. IEEE, Los Alamitos (1999)
7. Enslen, E., Hill, E., Pollock, L., Vijay-Shanker, K.: Mining source code to automatically split identifiers for software analysis. In: 6th IEEE International Working Conference on Mining Software Repositories, pp. 71–80. IEEE, Los Alamitos (2009)
8. Feild, H., Lawrie, D., Binkley, D.: An empirical comparison of techniques for extracting concept abbreviations from identifiers. In: Proc. of Int'l Conf. on Software Engineering and Applications (2006)
9. Høst, E.W., Østvold, B.M.: The Java Programmer's Phrase Book. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 322–341. Springer, Heidelberg (2009)
10. Høst, E.W., Østvold, B.M.: Debugging method names. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 294–317. Springer, Heidelberg (2009)

¹⁰ <http://oro.open.ac.uk/28352/>

11. Kuhn, A., Ducasse, S., Gírba, T.: Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49(3), 230–243 (2007)
12. Lawrie, D., Feild, H., Binkley, D.: Quantifying identifier quality: an analysis of trends. *Empirical Software Engineering* 12(4), 359–388 (2007)
13. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory* 10(8), 707–710 (1966)
14. Ma, H., Amor, R., Tempero, E.: Indexing the Java API using source code. In: 19th Australian Conf. on Software Engineering, pp. 451–460 (March 2008)
15. Madani, N., Guerrouj, L., Penta, M.D., Guéhéneuc, Y.G., Antoniol, G.: Recognizing words from source code identifiers using speech recognition techniques. In: Proc. of the Conf. on Software Maintenance and Reengineering, pp. 69–78. IEEE, Los Alamitos (2010)
16. Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., Sergejev, A.: Static techniques for concept location in object-oriented code. In: Proc. 13th Int'l Workshop on Program Comprehension, pp. 33–42. IEEE, Los Alamitos (2005)
17. Rațiu, D., Feilkas, M., Jürjens, J.: Extracting domain ontologies from domain specific apis. In: Proc. of the 12th European Conf. on Software Maintenance and Reengineering, pp. 203–212. IEEE Computer Society, Los Alamitos (2008)
18. Singer, J., Kirkham, C.: Exploiting the correspondence between micro patterns and class names. In: Int'l Working Conf. on Source Code Analysis and Manipulation, pp. 67–76. IEEE, Los Alamitos (2008)
19. Sun Microsystems: Code conventions for the Java programming language (1999), <http://java.sun.com/docs/codeconv>
20. Vermeulen, A., Ambler, S.W., Bumgardner, G., Metz, E., Misfeldt, T., Shur, J., Thompson, P.: *The Elements of Java Style*. Cambridge University Press, Cambridge (2000)

Revisiting Information Hiding: Reflections on Classical and Nonclassical Modularity

Klaus Ostermann, Paolo G. Giarrusso, Christian Kästner, and Tillmann Rendel

University of Marburg, Germany

Abstract. What is modularity? Which kind of modularity should developers strive for? Despite decades of research on modularity, these basic questions have no definite answer. We submit that the common understanding of modularity, and in particular its notion of information hiding, is deeply rooted in classical logic. We analyze how classical modularity, based on classical logic, fails to address the needs of developers of large software systems, and encourage researchers to explore alternative visions of modularity, based on nonclassical logics, and henceforth called *nonclassical modularity*.

1 Introduction

Modularity has been an important goal for software engineers and programming language designers, and over the last decades much research has provided modularity mechanisms for different kinds of software artifacts. But despite significant advances in the theory and practice of modularity, the actual goal of modularity is not clear, and in fact different communities have quite different visions in this regard. On the one hand, there is a classical notion of modularity grounded in information hiding, which manifests itself in modularization mechanisms such as procedural/functional abstraction and abstract data types. On the other hand, there are novel (and not so novel) notions of modularity that emphasize extensibility and separation of concerns at the expense of information hiding, such as program decompositions using inheritance, reflection, exception handling, aspect-oriented programming, or mutable state and aliasing, all of which may lead to dependencies between modules that are not visible in their interfaces.

This work is an attempt to better understand the relation between these different approaches to modularity by relating them to logic. *Classical logic* is the school of logic prevalent in modern mathematics, most notably first-order predicate logic. We argue that the “modularity = information hiding” point of view is rooted in classical logic, and we illustrate that many of the modularity problems we face can be interpreted in a novel way through this connection, since the limitations of classical logic as a representation formalism for human knowledge are well-known. This is in stark contrast to the programming research community, in which information hiding is nowadays such an undisputed dogma

of modularity that Fred Brooks even felt that he had to apologize to Parnas for questioning it [9]. Our analysis of information hiding in terms of classical logic suggests that there are good reasons to rethink this dogma.

To make one thing clear upfront: We do of course *not* propose to abandon information hiding or modularity in general; rather, we suggest to investigate different notions of information hiding (and corresponding module systems), inspired by nonclassical logics, that align better with how humans structure and reason about knowledge.

Since there is no precise definition of modularity available, we will use the following working definition in the beginning: Modularity denotes the degree to which a system is composed of independent parts, whereby ‘independent’ implies reusability, separate understandability and so forth.

A concern is *separated* if its code is localized in a single component of a system, such as a file, a class, or a container. *Information hiding* denotes the distinction between the *interface* of a software component and its *implementation*. The interface specification should be weaker than the implementation so that an interface allows multiple possible implementations and hence leaves room for evolution that does not invalidate the interface. The interface specification being weaker also means that an implementation can have multiple interfaces at the same time; in particular, one can talk about the interface of a module to another module as its weakest interface necessary to satisfy the other module’s needs [62]. An interface is also an *abstraction* of the implementation because it does not only hide (parts of) the implementation but also abstracts over it, that is, it allows reasoning on a more abstract level. For instance, rather than describing partial details of a sorting algorithm, it just states that the result is a sorted list.

A key question in information hiding is which information to hide and which information to expose. Parnas suggested the heuristic to hide what is ‘likely to change’ [58].

Modularity can also be viewed from the technical perspective of module constructs in programming languages. Module constructs typically enforce desirable properties such as separate compilation through type systems or other restrictions and analyses. While we appreciate the numerous wonderful works on module constructs, in this paper we want to discuss the more general question of how to organize, decompose, and reason about complex software systems, which is more basic than the question of how to enforce a given decomposition discipline by module constructs.

In the remainder of this paper, we formulate and defend the following five hypotheses:

1. The modularity and abstraction mechanisms that we use today are in deep ways tied to classical logic (and henceforth called *classical modularity* in the remainder of this paper; Sec. 2).
2. Classical modularity mechanisms and reasoning frameworks do often not align with how programmers reason about their programs (Sec. 3).
3. Successful information hiding is limited by the degree of separation of concerns, the inherent complexity of the system, and the need to support software evolution (Sec. 4).

4. The explanation for these problems is that programs are not like abstract, idealized scientific models – an analogy that has shaped the understanding of modeling in software development – but rather complex real-world systems (Sec. 5).
5. To overcome these problems, we have to weaken the assumptions of classical modularity and investigate notions of modularity based on nonclassical logics. Some existing attempts to escape classical modularity can be understood as being based on nonclassical logics (Sec. 6).

We conclude the paper with a proposal for a novel definition of modularity that makes the connection between a program and the logic in which we reason about its properties explicit.

2 Modularity and Classical Logic

The classical understanding of modularity is highly related to (and possibly shaped by) classical logic, and therefore, the basic principles and limitations of classical logic are relevant for modularity, too.

The spirit of classical logic is captured by the following quote from Lakatos:

The ideal theory is a deductive system with an indubitable truth injection at the top (a finite conjunction of axioms) — so that truth, flowing down from the top through the safe truth-preserving channels of valid inferences, inundates the whole system. [37]

This view of logic and proofs can be distilled into a number of basic principles, such as

- *The principle of explosion*: Everything follows from a contradiction.
- *Monotonicity of entailment*: If a statement is proven true, its truth cannot be renounced by adding more axioms to the theory, because *proofs are for eternity*, and if we learn more, we do not have to revise earlier conclusions.
- *Idempotency of entailment*: A hypothesis can be used many times in a proof.

and also a number of “nonprinciples” or “don’ts”, such as:

- *Inductive reasoning* – generalizing from examples – is unsound.
- *Reasoning by defaults* (such as “typically birds can fly”) or *Occam’s razor* (prefer the simplest explanation) is unsound.
- *Closed-world reasoning*, such as drawing conclusions by searching the knowledge database, is unsound.

These basic principles are common to all classical logics [21], that is, the logics most commonly used in mathematical reasoning since Frege’s *Begriffsschrift*, most notably first-order predicate logic.

Although, nowadays, these and similar properties are often taken for granted, they are actually specific to classical logics. Nonclassical logics do not have all of

these properties, or may allow reasoning using one of the aforementioned “non-principles”. For instance, paraconsistent logics give up the principle of explosion, that is, a contradiction has only “local” consequences and does not render the whole theory trivial. Another example are nonmonotonic logics, which give up the monotonicity of entailment. A well-known proof rule which is nonmonotonic is negation-as-failure [11], which means that a proposition is considered true if its negation cannot be proven – an example of closed-world reasoning. A well-known logic that gives up idempotency of entailment and monotonicity is linear logic [22].

A fundamental concept in logic is the distinction between proof theory and model theory [27]: For a set of axioms A formulated in the logic we can, via the syntactic deduction rules of the proof theory of that logic, prove theorems $A \vdash T$. On the other hand, we have the semantic notion of a *model* or *structure* M of a set of axioms, which is a mathematical structure that satisfies all axioms: $M \models \phi$ for all $\phi \in A$ using an interpretation function that assigns mathematical objects to the symbols occurring in the axioms. These semantic and syntactic views are typically related by soundness and completeness theorems. A soundness theorem says that all theorems that can be deduced from the axioms (the *theory* of the axioms) hold in all models of the axioms. The completeness theorem says that every theorem that holds in all models can also be deduced.

How is this related to modularity? In the following, we discuss some principles that we believe to constitute our understanding of (classical) modularity as information hiding, and relate them to classical logic as described above. We do not claim that all these principles have necessarily been shaped with classical logic in mind, but we believe that classical logic is the best formalization of the notion of abstraction that connects all these principles.

2.1 Information Hiding and Abstraction

Information hiding is to distinguish the concrete implementation of a software component and its more abstract interface, so that details of the implementation are hidden behind the interface. This supports modular reasoning and independent evolution of the “hidden parts” of a component [58]. If developers have carefully chosen to hide those parts ‘most likely to change’ [58], most changes have only local effects: The interfaces act as a kind of firewall that prevents the propagation of change.

Abstraction can be seen as a different take on information hiding, focusing more on the removal of information and the generalization of concrete to parameterized components that can be instantiated again. This includes the idea of having more than one instantiation of the same abstract component at the same time, thereby promoting code reuse.¹

Both information hiding and abstraction imply some notion of *substitutability*: A module’s implementation can be replaced by a different implementation

¹ Parnas and his colleagues have often used the word *abstract interface* for what we call just *interface* [60,8]; in Parnas’ terminology, an interface is between two software modules and describes the assumptions one module makes about the other.

adhering to the same interface, and since the implementation was hidden to other components in the system in the first place, these other components should not be disturbed by the change. Parnas was one of the first researchers to investigate this influence of information hiding on software evolution [58,63], but the idea shows up in many different forms:

- In structured programming, control structures such as loops hide the details of control flow management. Compilers are then free to choose among different implementations of the control structure with low-level jumps.
- Procedural and functional abstraction hide the implementation of an algorithm behind a procedure or function signature/contract. A procedure or function can then be replaced by a different implementation of the same contract.
- In object-oriented programming, encapsulation can be used to achieve information hiding². Objects of a class can then be replaced by objects of a subclass. The *Liskov substitution principle* [41] codifies this idea: The instances of a subclass should have the same observable behavior as the instances of the superclass when observed through the interface of the superclass, so that substitution of a subclass instance for a superclass instance does not change the observable behavior of the overall program.
- Data abstraction mechanisms hide the internal representation of an abstract data type, for instance, whether a complex number is stored in polar or Cartesian coordinates [67]. Logically, abstract data types are a form of existential quantification [48]. The internal representation of an abstract data type can be replaced with a different representation type supporting the same interface. Reynolds formalized and proved this property of abstract data types in his *abstraction theorem* [67].

The distinction between an interface and implementations of that interface, which is the at the core of information hiding and abstraction, is related to logic. The interface corresponds to a *set of axioms*, and the implementation of the interface corresponds to a *model of the axioms*. Substitutability is reflected by the fact that the same theorems hold for all models of the axioms (by soundness of the logic), hence we cannot distinguish two different models within the theory. The heuristic of hiding what is most likely to change is reflected by the design of axiom systems (say, the axioms of a group in abstract algebra) in such a way that there are many interesting models of the axioms.

2.2 Reductionism and Compositionality

Reductionism is the belief that a complex system can be understood completely by understanding its parts and the rules with which they are composed. This very general idea is not limited to software systems, and it has been described many times in the history of sciences, for instance, by Descartes [14]. A more recent take by Dawkins [13] describes *hierarchical reductionism* as the idea that

² Encapsulation is a somewhat ambiguous term. We follow Booch's definition [5] here.

complex systems can be described with a hierarchy of organizations, each of which is only described in terms of objects one level down in the hierarchy. For instance, a computer can be explained in terms of the operation of hard drives, processors, and memory, but it is not necessary to talk about logical gates, or even about electrons in a semiconductor. It is not surprising that this idea has been picked up and advocated in programming once program size became an issue [16,59].

Reductionism is an implicit assumption underlying classical modularity: When analyzing a modular software system, we want to understand it in terms of our understanding of the modules that constitute the system [63].

In the context of language semantics, the ideas of reductionism have been formally stated as *compositionality*. A semantics is compositional if the meaning of a complex expression is fully determined by the *meanings* of its constituent expressions and the rules used to combine them, rather than by the constituent expressions themselves. Compositionality is deeply grounded in mathematics through its relation to the notions of structure-preserving mappings, that is, homomorphisms and morphisms in universal algebra and category theory respectively, since a compositional function preserves the structure of its argument, and conversely a structure-preserving mapping is compositional [49].

As in the case of information hiding and abstraction, compositionality implies a strong notion of substitutability: If a subprogram is substituted by a different subprogram with the same meaning, the meaning of the whole program will still be the same. In other words, we can successfully reason more abstractly on an expression by thinking of its meaning rather than of the expression itself. When reasoning about the program, we can identify expressions having the same meaning. This process is typically called *equational reasoning*. Since the actual expression is hidden behind its meaning, compositionality can also be seen as a specific form of information hiding by considering the meaning of a program to be its interface.

Classical logic reflects the ideas of compositionality and reductionism in two ways: First, classical logic is compositional in the sense that a subset of the axioms of the theory can be exchanged by other axioms if they are logically equivalent (which means that they have the same deductive closure) without changing the set of theorems that hold for the whole set of axioms. Second, the “meaning function” (such as: determining whether a formula holds in a specific model) of classical logic is also compositional, meaning that the truth value of a composite formula is determined by the truth values of its constituents.

In the field of programming, compositionality is the hallmark of denotational semantics [78] and initial algebra semantics [23]. The denotation of a program is an *abstraction* of the program: different programs (such as $1+2$ and 3) have the same denotation (such as the mathematical object 3). In this sense, the denotation can be understood as an abstraction of the program, or, conversely, a program can be understood to be a model of its denotation. This may sound somewhat as if it is the other way around, since the denotation function maps syntax to semantics, but it makes sense if we consider the program to stand for

its “actual denotation” when executed on physical hardware. The “actual denotations” of $1+2$ and 3 are clearly different. They differ, for instance, in their power consumption or heat production of the CPU and required runtime. The difference between denotation and “actual denotation” hints at a principal limitation of compositionality: Those aspects of the “actual denotation” of a program that are abstracted over in its denotation may, to some user, be just as important as those that are reflected by its denotation. To take a practical example, whether a compiler of a programming language performs tail-call optimization [76] or not will often determine whether a program can be executed successfully or terminate with a stack overflow error. One can of course always enrich the semantic domain by more elements of the “actual denotation” (or abstractions of the “actual denotation”, such as a specification of the space behavior of procedure calls [12]), but it is not clear when to stop enriching the domains, since different stakeholders need to work with different equivalence classes of programs.

2.3 Idealization

The notion of idealization can be traced back at least to Plato and his idea of *ideas* or *forms* [71]. He holds that there are abstract notions – ideas – that capture the essence of aspects of our real life, yet never actually occur in real life. For instance, there are no perfect circles in real life, yet we can talk about the idea of a perfect circle.

Idealization was used systematically by Galileo, who, in his study of bodies in motion, made assumptions such as frictionless surfaces and spheres of perfect roundness. The motivation for idealization is that actual scientific objects are too complicated, hence they need to be summarized to a few properties relevant to the phenomenon under study.

In the computer science community, Dijkstra motivates idealization from a modularity perspective as follows:

A scientific discipline separates a fraction of human knowledge from the rest: we have to do so, because, compared with what could be known, we have very, very small heads. [17]

Modularization and idealization are hence rather similar ideas: To deal with complexity by being able to concentrate on those things that are relevant to the task at hand and to ignore the rest for the time being. Idealization is an implicit assumption underlying modularity: Our understanding (or, the interface) of a module is an idealization of the actual implementation of the module. Interfaces are an idealization in the sense that they assume that some aspects of the implementation are not relevant (such as whether a display update is triggered when calling an interface function), which may lead to false assumptions about the implementations of the modules [33].

The axiomatic method of classical logic described in the beginning of this section can be seen as a formalization of idealization, where the axioms play the role of an idea, and its models are the real-world objects captured by that idea.

2.4 Monotonicity

Monotonicity is the idea that we want to prove things “once and for all”. It means that we never have to withdraw a conclusion when we learn more. For instance, if we establish a property of a software system in a monotonic logic, we never have to revise that property when more components are added to the system. As described above, monotonicity is one of the defining properties of classical logics.

Program logics such as Hoare logic [28] are typically monotonic: Enlarging the program does not invalidate what was proved about the contained smaller program. Also, operational models of programming languages can in most cases be considered a monotonic logic in the following sense: If we consider the equational theory implied by the operational semantics of the language, then typically we have the property that if $e = e'$, then $E[e] = E[e']$ for an evaluation context E that plugs the expression into a bigger program [50,81]. This congruence property allows us to reason about the behavior of programs in a modular way: We do not have to revise our conclusions about program behavior when we enlarge the program or use it as subprogram in a bigger program.

2.5 Summary

The common notion of modularity, especially the facet of information hiding, is deeply related to classical logic. We take compositionality of abstractions and monotonicity in reasoning about them for granted. Classical logic shapes our thinking and expectation of modularity. However, as we will argue next, humans (and hence programmers) do not always organize and reason about knowledge in accordance with classical logic, which threatens the implicit assumption of classical modularity, namely that information hiding in the strong sense presented here is the best means to deal with software complexity.

3 Programmers Use Nonclassical Reasoning

Although our modularity mechanisms are shaped by classical logic, programmers frequently reason about software systems in nonclassical logics. Programmers use *inductive reasoning*, use *default reasoning and Occam’s razor*, and use *negation-as-failure and closed-world reasoning*, as we will illustrate. All these means of reasoning are unsound from the perspective of classical logic (and have been formalized in various nonclassical logics), but are still used in everyday development and maintenance tasks. Hence, classical modularity mechanisms frequently do not support programmers adequately when reasoning about their programs.

3.1 Programmers Use Inductive Reasoning

Programmers routinely infer a general software property from observing individual cases. For example, from a lack of bugs in specific cases, developers tentatively infer the lack of bugs of a software. This is the essence of testing. From

the perspective of classical logic, however, a successful test case shows nothing; only a failed test case produces new knowledge.

Similarly, developers sometimes explore the behavior of a software module by testing it on some inputs, and infer, through inductive reasoning, general laws on how the module behaves, especially when its APIs are underspecified. Alternatively, they might know the behavior of a module A only partially, use it to build module B , and later learn details about the behavior of A (for instance, corner cases) by testing the complete software.

The success of tools like Daikon [19] or the technique from Henkel and Diwan [26], which discover likely program invariants or algebraic specifications by inductive reasoning over test case results, also illustrates that inductive reasoning over programs *does* produce useful knowledge.

We could argue that inductive reasoning is unsound and programmers should not use it, but there are good arguments to the contrary. First, all basic theories in natural sciences are essentially the product of inductive (or abductive) reasoning; all theories in natural sciences can only potentially be falsified, but never be proven correct [64]. In that sense, inductive reasoning has a quite impressive track record. Second, inductive reasoning is natural human behavior. This hypothesis is supported by the basic learning mechanism of the human brain at the neuronal level, known as Hebbian learning [25]: Our brain learns correlation between different concepts and expects that this correlation will repeat in the future. Conditioned reflexes are a prime example of such learning process. Recent advances in computational neuroscience provide models that successfully explain many higher-level behaviors through the basic mechanism of Hebbian learning [47]. Detailed studies are available mostly for vision (for instance, illusory contours are explained this way), but brain processing uses the same fundamental processing mechanisms for all kinds of information; thus, researchers conjecture that all brain functions might be explained in terms of associative learning.

3.2 Programmers Use Default Reasoning and Occam's Razor

Programmers tend to use the simplest explanation they can imagine for an experienced phenomenon, such as a bug. Similarly, they tend to predict the simplest behavior consistent with the interface for a software entity, such as an API. Nevertheless, they regard such explanations and predictions as tentative, that is, programmers infer them nonmonotonically and revise them when contradicting evidence is discovered.

Default reasoning and Occam's razor are common in everyday development tasks; consider the following examples: (1) Developers might assume that an API function will not perform the side effect of formatting the harddrive or modifying the value of the provided arguments unless it is explicitly specified. (2) When a program terminates by printing "NullPointerException", developers typically assume a raised `NullPointerException` as cause, rather than a `println("NullPointerException")` instruction in the program. (3) Developers may expect that getter methods do not mutate the receiver object and can thus

be safely invoked in a read-only fashion from multiple threads. (4) Developers might observe patterns in an API from a few of its members, and use such inferred patterns as default rules. Default reasoning is also acknowledged by design principles such as the *principle of least astonishment* [4], which recommends that the API should not contradict common predictions of the programmer.

Of course, these reasoning patterns can lead to invalid results when additional observations are made. This is actually quite common and may be the cause for some debugging efforts. For example, the second author shared the third assumption about thread-safe getter methods and had to revise his reasoning in a program using a well-known open-source library.

Again, we could put the blame for false preliminary conclusions on the programmer, but Occam’s razor and default reasoning appear to be “hard-wired” human behavior, as also supported by the law of *prägnanz* in Gestalt psychology, which says that we tend to order our experience in a way which is regular, orderly, symmetric and maximizes simplicity [77].

3.3 Programmers Use Negation as Failure and Closed-World Reasoning

Programmers often reason about a closed code base. For example, when removing a method that is presumably no longer necessary, they confirm that the method is actually no longer necessary by checking whether this method is still called in the current code base. These kinds of API changes are of course avoided if possible, if the API is used by a large number of applications outside the control of the programmer or company, but it is well-known that this means that APIs often become a kind of “software asbestos” [35,3] or leads to versioning problems such as the infamous ‘DLL hell’, if the API change cannot be avoided.

Reasoning about callers of a method is an example of the negation-as-failure proof rule, which allows to deduce a property $\neg P$ from failure of proving P [11] (for instance, with $P = \text{“method foo is used”}$). It is an example of closed-world reasoning [65] as well, because such (nonmonotonic) reasoning might be invalidated when the considered scenario is extended to elements allowing to prove P . Negation as failure and closed-world reasoning are both incompatible with classical logic.

Conflicts between classical logic and closed-world reasoning frequently arise during software evolution, especially in the context of APIs. Stable APIs are very difficult to achieve in open systems that might be extended by others (it is essentially impossible to ever remove any API functionality). Therefore, many developers are less strict about stability and information hiding and tend toward a closed-world assumption. For example, the Linux kernel developers do not guarantee stable APIs and instead strongly urge maintainers of external code to submit that code for inclusion in the kernel, so it can be reasoned about and evolved together with the APIs in a closed-world fashion [36].

Closed-world reasoning is also required to establish many other important properties of software, for instance, temporal or concurrency properties.

This holds for informal manual reasoning, but is even more obvious when one considers automated tools such as model checkers and static analyses, which assume a closed world when reasoning about source code. A property established in one code base may no longer hold in a larger versions of the program. For instance, suppose a module acquires lock A and then lock B (while holding lock A) and model checking ensures this module is safe; suppose then a new module is introduced, acquiring locks A and B in the reverse order. As we know, this will cause a deadlock, which will affect also the existing module. These limitations are of course well-known in these communities (e.g., [140]); we mention them here to support our point that programmers and their tools routinely and successfully used inductive reasoning.

3.4 Discussion

We have shown that in many cases programmers use reasoning that does not align with classical logic and which causes problems in the context of classical modularity. One could ask, whether we should blame programmers or the modularity mechanisms. For example, we could blame programmers, because they are presumably just too lazy to use proper reasoning, or we could blame modularity mechanisms, because they do not support programmers adequately.

We take sides with the programmers for two reasons: First, we have evidence that various patterns of nonclassical reasoning are “hard-wired” into human intelligence. It seems natural to us that programming methodology should embrace rather than denunciate the way of reasoning humans are born with. Second, many important properties of programs just cannot be established using classical reasoning: Viewed as axioms of logical theories, module interfaces are highly incomplete, that is, for many propositions P neither P nor $\neg P$ can be proven classically (such as in the examples from above: $P = \text{“method foo is used”}$, or $P = \text{“the program is deadlock-free”}$). Hence, programmers are essentially *forced* to use nonclassical reasoning, because there is no way to prove or reject many relevant properties with classical reasoning.

These two reasons suggests that the power of classical “modular reasoning” is rather limited, because it only works for so few properties. We believe that modularity mechanisms should be adapted accordingly to better reflect how programmers reason about code. In light of this finding, the next section will analyze the limitations of classical information hiding in detail.

4 Limits of Information Hiding

Information hiding is typically regarded as a core achievement and goal of modularity in the struggle to reduce complexity [63]. However, programmers often experience limitations where information hiding is difficult or does not seem to pay off. In the following, we describe some situations where the limitations of information hiding become apparent.

4.1 Operational Behavior and Interface Detail

If a stakeholder wants to reason about “nonfunctional”³ aspects of a system, such as time or space complexity or power consumption, he probably needs to reason about implementation details hidden behind abstraction barriers.

For example, when hiding the representation of complex numbers as either Cartesian or polar coordinates [67], the choice of representation is irrelevant from the perspective of Reynold’s abstraction theorem, as already discussed in Section 2.1. However, the implementation choice makes a difference when executing the program on physical hardware. For example, different implementations have different time or space behavior of the operations, different rounding errors, different optimizations that the compiler will apply, or different power consumption. To some stakeholders, such concerns may well be important; while some require higher performance, others require higher precision.

To support the information needs of such a stakeholder, one could expose performance and precision information, for example, by adding additional constraints to the interface of the complex-number data type. But with each additional constraint, the possible implementations are more and more limited, until eventually all information is exposed, and just one possible implementation remains. By strengthening the interface, the distinction between interface and implementation is weakened, and information hiding is rendered useless. In logic, this situation is formalized by the notion of *completeness*, which denotes a logical theory that has just one model (up to isomorphism).

In a modular structure based on a nonclassical form of information hiding, it may be possible to establish additional interface properties by nonclassical (e.g., inductive or default) reasoning, without explicitly stating all of them in the interface. A concrete example would be a default rule which says that “getter” methods usually do not perform side-effects. An example of inductive reasoning would be to observe that many functions of an API that access the file system are not thread-safe, and generalize this finding to all API functions that access the file system. In the field of logic, the problem of having to state too many properties explicitly to reason about an ‘API’ is known as the *qualification problem*, which we will discuss in more detail in Sec. 4.5.

4.2 Large Systems

When information is hidden behind an abstraction barrier, there are potential stakeholders (or concerns), who are interested in that hidden information. Therefore, the success of information hiding depends on whether such potential stakeholders (or concerns) and their information needs are relevant for the system or not – and, the larger the software system, the more likely a relevant stakeholder exists. In that sense, we argue that strict information hiding is problematic in

³ Actually the word *nonfunctional* is a misnomer, since nonfunctional properties are just as important aspects of the function of a system as its “functional” properties. The wording is unfortunate, because it is an excuse to pretend that some aspects of a system can be ignored when “modeling” a system, see also the discussion in Sec. 5.

large systems. This has strong implications on practice, as we exemplify with the Linux kernel.

Surrounding the origins of the Linux kernel there is a well-known debate about how to design an operating system kernel between Linus Torvalds, the original developer behind Linux, and Andrew Tanenbaum, an operating system researcher [15]. At the heart of the debate lies another debate about modularity. Academics argued that kernels of operating systems should be written using loosely coupled independent modules and that interface boundaries should be enforced through shared-nothing, message-passing-based concurrency (known as microkernel design [79]). In contrast, Linux uses a monolithic kernel design which does not enforce information hiding strictly.

In a nutshell, Torvalds' motivation for neglecting information hiding is that parts of the kernel are highly interdependent.⁴ They require so much knowledge about each others implementation that there remains little to hide. Torvalds himself provides (among many others) the following example:

This is an example of how things [different modules] are *not* “independent”. The filesystems depend on the VM [Virtual Memory subsystem], and the VM depends on the filesystem. You can't just split them up as if they were two separate things (or rather: you *can* split them up, but they still very much need to know about each other in very intimate ways)⁵

So, programmers of the Linux kernel, and in fact of most other operating systems as well, accept a weaker form of modularity because so much information would have to be exposed in interfaces that information hiding does no longer add enough value.

A related issue of large systems arises with crosscutting concerns such as transactions or concurrency. The problem that such concerns are very hard to modularize with classical modularity mechanisms has been the motivation for aspect-oriented programming [32].⁶ If such concerns are *not* modularized, however, a basic assumption of information hiding, namely monotonicity, does not hold anymore: Composing two programs which are each separately correct with respect to, say, lock-based concurrency or transactions, are in general no longer correct when composed. More importantly, the noncomposability can in general not be deduced from the interfaces of these components (or it is at least not clear how to document the components in such a way that it is). Hence, the monotonicity assumption of classical modularity fails when concerns are not properly separated.

⁴ Actually, also performance is a common argument for monolithic kernels. Although some modularity mechanisms may arguably add some performance penalties, we ignore this aspect to concentrate on the issue at hand.

⁵ http://kt.earth.li/kernel-traffic/kt20050103_289.html#1

⁶ There is no consensus whether AOP solves these problems (e.g., [34]) but this is not relevant to our point.

4.3 Separation of Concerns and the Dominant Decomposition

When taking the point of view that what is hidden behind an interface is (or belongs to) a concern, it becomes obvious that better separation of concerns reduces the amount of information hiding in the system. For instance, in the canonical AOP example of updating a display when a figure element changes [33], a figure element module hides less information behind its interface when the display updating logic is separated from the figure element module. In that sense, and contrary to the common notion that information hiding and separation of concerns go hand in hand, information hiding and separation of concerns can actually be contradictory.

The *tyranny of the dominant decomposition* [80] also reflects a major limitation of information hiding: What can be hidden behind an interface depends on the chosen decomposition, but there is no “best” decomposition; rather, from each point of view (such as the points of views of the different stakeholders) a different decomposition (and hence information hiding policy) would be most appropriate. What one stakeholder would hide as an implementation detail behind an interface is of primary importance to another stakeholder, who would hence choose a different decomposition that exposes that information.

4.4 Software Evolution

Even if a software system is successfully modularized, and the information needs of all stakeholders and concerns are reflected in the interfaces of components, information hiding might still hinder software evolution. This might be surprising at first, because information hiding is supposed to facilitate software evolution by hiding design decisions behind interfaces, so that they can be changed at will. The problem is that the original developers have to anticipate change and to modularize the software accordingly.

Unfortunately, it is not clear how to decide up-front which design decisions need to be hidden and which need to be exposed. Parnas heuristic of hiding what is most likely to change is difficult to follow [7]. If a design decision is exposed in the interface of a component, this aspect of the component cannot be evolved in a modular fashion later. But if the design decision is hidden behind the interface, software evolution might bring a new stakeholder (or concern) into the system which needs to access that hidden information. So, to support the information need of this stakeholder (or concern), the design decision should not have been hidden in the first place.

An example for this situation is discussed in the aforementioned display-update example: When the `Point` figure element hides its update logic behind its interface, the system cannot evolve to support also a `Line` abstraction based on `Point`, since the update logic of `Line` cannot be implemented without detailed knowledge about the update logic of `Point` [33].

⁷ The wording ‘most likely’ indicates that one has to use nonclassical – such as inductive or probabilistic but in any case nonmonotonic – reasoning to determine the modular structure of a system. Hence the illusion of staying within classical logic all the way through breaks together one way or the other.

One could argue that successful modularization just needs better planning [61] to better assess what is likely to change, but we believe that this is an implausible assumption because large-scale software systems are assembled from many independently developed and independently evolving parts; hence, a big global “plan” is infeasible and unanticipated changes are unavoidable in long-living projects. In fact, the mere assumption of a monolithic global plan is contradictory to modularity.

4.5 Information Hiding and Classical Logic

As lesson, we infer from these examples that the larger and more complex a software system is, the harder a strict classical discipline of information hiding can be maintained. There are many concerns that, when separated, need to expose implementation detail in such a way that information hiding is impaired. Developers have to decide what information to hide and what to separate. This is a fundamental problem of classical modularity, which can be traced back to problems well-known in classical logic. For instance, the *qualification problem* describes the problem that

in order to fully represent the conditions for the successful performance of an action, an impractical and implausible number of qualifications would have to be included in the sentences expressing them [44].

McCarthy gives the following example:

The successful use of a boat to cross a river requires, if the boat is a rowboat, that the oars and rowlocks be present and unbroken, and that they fit each other. Many other qualifications can be added, making the rules for using a rowboat almost impossible to apply, and yet anyone will still be able to think of additional requirements not yet stated.

From the perspective of modularity, the qualification problem is clearly about information hiding, or more precisely, about the difficulty of information hiding in classical logic.

We believe that a possible solution can be to restrict the expectations of information hiding driven by classical logic (for instance, not to expect to prove program properties “once and for all”), and open us to less strict forms of information hiding, as we will discuss in Section 6.

5 Programs Are Not Models

What is the cause of the failures of classical modularity discussed in the previous two sections? We believe that the answer to this question lies in the notion of modeling and idealization from natural sciences (and, eventually from Plato’s ideas and Aristotle’s notion of *essence*), as discussed in Section 2. A scientific

model⁸ of a physical phenomenon removes information not relevant for the purpose of the model, and makes simplifying assumption to distill the core of the phenomenon the model is supposed to illustrate. It is not surprising that classical logic is a good match to describe natural scientific models, since historically, mathematics and logics were developed as auxiliary sciences to support natural science.

It seems tempting to assume that software is a model in that sense, too, and we believe that this is indeed a wide-ranging implicit assumption of many software researchers. Software engineering books talk about “modeling” all the time. There is even a branch of software engineering called “model-driven development”, in which the actual programs are explicitly called “models”. The Object Management Group defines: “An object models a real world entity” [52], and the point of view that programs should model the real world has been quite important in Simula and the whole Scandinavian tradition to OO programming [42].

However, large software systems are not like that. They have to take into account the desires and needs of many different stakeholders. They have to deal and interact with the real world, which means that simplifying assumptions often turn out to be false. Instead of being like a scientific model, software systems are more like a mix of many overlapping and interacting models. One could of course say that a mix of overlapping models is another, more complicated model. But we believe that it is no longer useful to consider big programs to be models of a part of reality, but rather to *be* a part of reality. For instance, while at some early stage the programming concept of an order may have been a model of a hand-written document in a company, there are nowadays typically no artefacts beyond the record in the database that represent the order: It *is* the order. In contrast to a natural science model, a program is not describing a physical phenomenon – it *is*, when running, a physical phenomenon.

In natural sciences the problems of idealization and abstract models are well-known, of course. For instance, when trying to compute the movements of actual bodies in motion, aspects such as friction or air resistance have to be taken into account – the simplifying assumptions do not hold anymore. Taking all these additional influences into account turns Galileo’s simple models into highly complex computations. Even in natural sciences itself, there is discussion about whether scientific models are really accurate descriptions of physical phenomena [10], and a process of de-idealization and de-simplification is proposed to turn the model into an accurate description of reality [46,39].

The problem of multiple overlapping models, which manifests itself as the tyranny of the dominant decomposition in software (cf. Sec. 4.3), is also well-known in natural scientific modeling:

All of our theories and models are tightened together only because they apply to the same empirical reality but do not enter into any further relations (deductive or otherwise). We are confronted with a patchwork

⁸ In this section we use the term *model* to denote *scientific model* and not as the term is used in logic, which is confusingly different.

of theories and models, all of which hold *ceteris paribus* in their specific domains of applicability. [24]

Complex, de-idealized patchworks of scientific models, as required for simulations of the real world, are much more akin to large software systems, since both have to deal with many aspects of reality and do not have the luxury to abstract over aspects that are inconvenient for a simple, elegant model.

The misleading analogy between programs and natural scientific models explains the failure of information hiding, since classical logic – the foundation of information hiding – is the framework in which scientific models are implicitly or explicitly formulated.

6 Towards Nonclassical Modularity

In the beginning of Sec. II, we have pointed out several modularity mechanisms that can be used to improve extensibility or separation of concerns, but have been criticized for restricting information hiding and modular reasoning, for instance inheritance [75], aspect-oriented programming [2], reflection, aliasing and mutation [51,54], multithreading [20], and exception handling [69].

Many of these modularity mechanisms can be understood to leave the “safe” world of classical logic, and indeed they can be understood to correspond to different nonclassical logics. Here are a few examples.

Aspect-Oriented Programming and Reflection. In earlier work, the first author has shown that aspect-oriented programming can be understood in terms of a nonmonotonic logic called *default logic* [56]. The idea is that one can reason by default that the semantics of a method call is to execute the corresponding method body, similar to how classes themselves can be interpreted as defining a default behavior that may be refined by subclasses (see discussion of inheritance below). Aspects that intercept such method calls are considered exceptions to that default rule. Hence, in this setting, one can – using defaults – reason locally about the program behavior. In case one learns later that the default assumption turns out to be wrong, there is a controlled process of updating the conclusions one has drawn from the invalid default assumption [56]. Using the logic proposed in this paper, one can establish a property such as “display updating is consistently applied when the data changes” modularly, by only considering the aspect that maintains this property.

Reflection (e.g., [74]) is also known to be a powerful modularity mechanism (e.g., [31]), but is in conflict with information hiding, since implementation details of foreign modules can be observed and modified. Not surprisingly, reflection is also frowned upon in classical logic ever since the paradoxes of naive set theory (Russel paradox, Cantor paradox, etc.) have been discovered – all of which rely on a form of reflection, namely self-application.

Aliasing and Mutation. The program-verification community has developed separation logic [68] to reason about programs using pointers, aliasing, etc., in a

modular way. Separation logic is a nonclassical logic, since the structural rules of weakening (monotonicity of entailment) and contraction (idempotency of entailment) do not hold [53]. Instead, the so-called *frame rule* allows the programmer to reason about each routine separately, given that the parts of the heap that are modified by each routine are disjoint [68]. The frame rule solves a particular instance of the *frame problem* [45], which has been a major motivation for the development of many nonclassical logics and is at the same time a typical modularity problem, namely how to specify what a module *does not* do, without enumerating all possibilities [6].

Separation logic seems to be compatible with classical information hiding at first. However, the frame rule forces one to make all sharing and aliasing in the program explicit in the specification, which is contrary to the idea of using *implicit* communication via shared variables to reduce coupling and hence improve modularity [18, Sec. 2] [72, Sec. 4.8.2]. Specifying sharing and aliasing explicitly has a ripple effect, because typically the callers of the components that share variables have to know about this, hence the callers of the callers have to know, and so forth [72, Sec. 4.8.2], which means that the usage of separation logic in such cases becomes a form of closed-world reasoning. So one has only two choices: Either give up the modularity that can be gained by implicit communication, or use a stronger – unsound – form of the frame rule, similar to circumscription [44], that allows one to tentatively compose proofs of properties of program parts even if they do potentially communicate implicitly.

Inheritance. Ideas to understand classes in object-oriented languages as giving nonmonotonic default definitions that may be refined by subclasses are almost as old as object-oriented programming itself [66,73]. More recently, variants of separation logic (see above) have been proposed to reason about object-oriented programs [57]. These logics illustrate that inheritance is a nonclassical modularity mechanism.

Temporal Logics. Temporal logics, such as linear-time logic (LTL), are a common formalism to reason about temporal properties of programs, especially concurrent programs [43]. Temporal logics assume a closed world, which means that the whole program (or state machine) to be verified must be fully known, and results established for one program do not automatically hold for extensions of that program (nonmonotonicity) or compositions of multiple programs [1].

Closed-World Modularity. Some approaches embrace closed-world reasoning and instead focus on tool support to dealing with nonmodular systems. For example, *FEAT* helps to discover and document scattered concerns and can afterward support reasoning about the still-scattered concerns (a closed knowledge base) by providing navigation support [70]. *Virtual separation of concerns* [30] emphasize editable views on code and whole-program analysis to reason about scattered implementations of a concern instead of enforcing a separation into modules. Ideas of *effective views* [29] and *on-demand remodularization* [55] take this even a step further and actually rewrite the source code on demand to match the form of locality or information hiding that the programmer needs for a task.

Error Handling. Lanier remarked that software “breaks before it bends” [38]. That is, a single failure (such as a null-pointer access) in a minor part can cause inconsistencies in the whole program, just like a single inconsistency in a logical theory allows to prove every proposition (including contradictory ones). We believe this similarity is not accidental: Software inherits this property from the principle of explosion of classical logic. One of the common points is that both a software module and a logical theory require the perfect consistency (such as: freedom of bugs), even if software modules are in daily practice are rarely exempt from inconsistencies.

Interestingly, both in logic and in software different but similar means have been developed to deal with such explosions/crashes. In logic, the field of *paraconsistent logics* [7] deals with logics that are inconsistency-tolerant, that is, where one can still draw reasonable nontrivial conclusions even if there is an inconsistency in some part of the theory. This is similar in spirit to attempts in computer science to limit the effects of errors, such as null pointer errors or nontermination, that would otherwise destroy a running program immediately.

For instance, insulating faults by partitioning a software in different processes is a traditional best practice in the Unix culture, because it increases stability.

More recently, Martin Rinard and his group introduced the notion of *failure-oblivious computing* [69] whose idea is that applications should continue to produce reasonable results despite unexpected errors, and proposed innovative and even surprising techniques for doing so. These techniques are often met with resistance, because they change the local semantics of the program (for instance, by skipping loop iterations to improve performance). That is contrary to the spirit of classical modularity; however, the only alternative is the principle of explosion.

Nonstrict programming languages (such as Haskell) can also be understood to restrict the propagation of a common error, namely nontermination. The connection to paraconsistent logics becomes particularly obvious when one identifies inconsistency with nontermination, which is also suggested by the fact that the same symbol, \perp , is used to denote both inconsistency in logic and nontermination in denotational semantics.

Discussion. While the results discussed in this section are rather preliminary and require a more formal investigation, we still consider it striking that many program structuring mechanisms that have been criticized for violating information hiding are at the same time similar to developments in nonclassical logics. Also, the motivations for these developments are often similar as well, for instance, avoiding the propagation of errors for both error recovery mechanisms and paraconsistent logics, or avoiding an excessive number of qualifications for both aspect-oriented programming and nonmonotonic logics (cf. Sec. 4.5).

We believe that these similarities indicate that the programming community should acknowledge that programs are a form of *knowledge representation*, and the same considerations with regard to modularity, extensibility, ease of reasoning, and so forth, apply to both logics and programs. Until now, programming languages have usually been developed independently of logics, and logics to

reason about various properties of the program have only been added as an afterthought. We believe that there is a lot of potential in the idea to make use of the connection between programming on one hand and logics and knowledge representation on the other hand, and develop modularity constructs and their logic side by side instead.

7 Conclusions

The traditional point of view on modularity as information hiding is deeply rooted in classical logic and inherits both its merits and limitations. As a device to structure the knowledge embodied by large-scale software system it is problematic, since there is a deep mismatch between the idealizing form of modeling for which classical logic was designed, and the multi-stakeholder reality of complex software systems. Some existing ideas to escape the limitations of traditional information hiding can be understood as being based on nonclassical logics. We propose to turn this observation into a principled design methodology for future modularity mechanism in which the modularity mechanism, its information hiding policy, and a corresponding (potentially nonclassical) logic are developed side by side. It does not make sense to judge a modularity mechanism through the glasses of a logic that does not match to the logic with which knowledge is organized in this modularity mechanism.

In fact, we believe that it is useful to adopt a novel definition of modularity⁹ that takes the relation between programs, its modules, and the logic we use to reason about the program into account.

Instead of talking about modularizing *concerns* – a term that has often been understood in a rather syntactic way – we propose to talk about modularizing *properties* of a program. Since the way we establish a property depends on the logic, modularity is also relative to the used logic L . Hence we can define property P to be modularized in a program unit U (which we assume to include its interface to the rest of the program) of a program, if P can be proved from U in L , or, using formal notation, $U \vdash_L P$.

Under this definition, cohesion denotes that a program unit modularizes a single (or few) coherent program properties. Program units are coupled, if an important property can only be proved from a larger set of program units, or even the whole program. A perfect (perhaps unattainable) modularization is one where all properties required by the specification are modularized.

We hope this definition will help to correct what we perceive to be a *modularity bias*: That some desired properties – such as the aforementioned “functional” properties of a system – are more important to modularize than other (“non-functional”) properties.

Acknowledgements. We particularly thank David L. Parnas for feedback, historical notes, and interesting discussions about an earlier draft of this paper. We also thank the reviewers for quite helpful comments. The authors of this work are supported by the ERC Starting Grant No. 203099.

⁹ This direction was actually suggested by an anonymous reviewer of this paper.

References

1. Abadi, M., Lamport, L.: Composing specifications. *ACM Trans. Program. Lang. Syst.* 15, 73–132 (1993)
2. Aldrich, J.: Open modules: Modular reasoning about advice. In: Gao, X.-X. (ed.) *ECOOP 2005*. LNCS, vol. 3586, pp. 144–168. Springer, Heidelberg (2005)
3. Bartolomei, T.T., Czarnecki, K., Lämmel, R., van der Storm, T.: Study of an API migration for two XML APIs. In: Malloy, B., Staab, S., van den Brand, M. (eds.) *SLE 2010*. LNCS, vol. 6563, Springer, Heidelberg (2011)
4. Bloch, J.: How to design a good API and why it matters. In: *Companion Int’l Conf. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, pp. 506–507. ACM, New York (2006)
5. Booch, G.: *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, Reading (2007)
6. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. *IEEE Trans. Softw. Eng.* 21, 785–798 (1995)
7. Bremer, M.: *An Introduction to Paraconsistent Logics*. Peter Lang Publishing (2005)
8. Britton, K.H., Parker, R.A., Parnas, D.L.: A procedure for designing abstract interfaces for device interface modules. In: *Proc. Int’l Conf. Software Engineering (ICSE)*, pp. 195–204. IEEE Press, Los Alamitos (1981)
9. Brooks, F.P.: The mythical man-month: After 20 years. *IEEE Software* 12, 57–60 (1995)
10. Cartwright, N.: *How the laws of physics lie*. Clarendon Press, Oxford (1983)
11. Clark, K.L.: Negation as failure. In: *Logic and Data Bases*, pp. 293–322 (1977)
12. Clinger, W.D.: Proper tail recursion and space efficiency. In: *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pp. 174–185. ACM, New York (1998)
13. Dawkins, R.: *The Blind Watchmaker*. Norton & Company, New York (1986)
14. Descartes, R.: *Discourse on the Method of Rightly Conducting One’s Reason and of Seeking Truth in the Sciences (1637)*, <http://www.gutenberg.org/etext/59>
15. DiBona, C., Ockman, S., Stone, M. (eds.): *Open Sources: Voices from the Open Source Revolution*. O’Reilly & Associates, Inc., Sebastopol (1999)
16. Dijkstra, E.W.: The structure of “THE”-multiprogramming system. In: *Proc. ACM Symposium on Operating System Principles*, pp. 10.1–10.6. ACM, New York (1967)
17. Dijkstra, E.W.: EWD 447: On the role of scientific thought. In: *Selected Writings on Computing: A Personal Perspective*, pp. 60–66. Springer, Heidelberg (1982)
18. Ernst, E.: Method mixins. In: *Proc. Net.ObjectDays/GSEM*, pp. 145–161. GI (2005)
19. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.* 27(2), 99–123 (2001)
20. Flanagan, C., Freund, S.N., Qadeer, S., Seshia, S.A.: Modular verification of multithreaded programs. *Theor. Comput. Sci.* 338, 153–183 (2005)
21. Gabbay, D.: Classical vs non-classical logic. In: *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 2, Oxford University Press, Oxford (1994)
22. Girard, J.-Y.: Linear logic. *Theoretical Computer Science* 50, 1–102 (1987)

23. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial algebra semantics and continuous algebras. *J. ACM* 24(1), 68–95 (1977)
24. Hartmann, S., Frigg, R.: Scientific models. In: Sarkar, S., Pfeifer, J. (eds.) *The Philosophy of Science: An Encyclopedia*, vol. 2, pp. 740–749. Routledge, New York (2005)
25. Hebb, D.: *The organization of behavior*. John Wiley & Sons, Chichester (1949)
26. Henkel, J., Diwan, A.: Discovering algebraic specifications from Java classes. In: Cardelli, L. (ed.) *ECOOP 2003. LNCS*, vol. 2743, pp. 431–456. Springer, Heidelberg (2003)
27. Hinman, P.: *Fundamentals of Mathematical Logic*. A K Peters, Wellesley (2005)
28. Hoare, C.A.R.: An axiomatic basis for computer programming. *Commun. ACM* 12(10), 576–580 (1969)
29. Janzen, D., De Volder, K.: Programming with crosscutting effective views. In: Vetta, A. (ed.) *ECOOP 2004. LNCS*, vol. 3086, pp. 197–220. Springer, Heidelberg (2004)
30. Kästner, C., Apel, S.: Virtual separation of concerns – A second chance for pre-processors. *Journal of Object Technology (JOT)* 8(6), 59–78 (2009)
31. Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.: Open implementation design guidelines. In: *Proc. Int’l Conf. Software Engineering (ICSE)*, pp. 481–490. ACM, New York (1997)
32. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Aksit, M., Auletta, V. (eds.) *ECOOP 1997. LNCS*, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
33. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: *ICSE*, pp. 49–58. ACM, New York (2005)
34. Kienzle, J., Liu, H.: AOP: Does It Make Sense? The Case of Concurrency and Failures. In: Deng, T. (ed.) *ECOOP 2002. LNCS*, vol. 2374, pp. 37–121. Springer, Heidelberg (2002)
35. Klusener, S., Lämmel, R., Verhoef, C.: Architectural Modifications to Deployed Software. *Science of Computer Programming* 54, 143–211 (2005)
36. Kroah-Hartman, G.: The Linux kernel driver interface, http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=blob;f=Documentation/stable_api_nonsense.txt
37. Lakatos, I.: A renaissance of empiricism in the recent philosophy of mathematics. *Br. J. Philos. Sci.* 27(3), 201–223 (1976)
38. Lanier, J.: One half of a manifesto: Why stupid software will save the future from neo-darwinian machines. *wired* 8.12 (2000)
39. Laymon, R.: Idealizations and the testing of theories by experimentation. In: Achinstein, P., Hannaway, O. (eds.) *Observation Experiment and Hypothesis in Modern Physical Science*, pp. 147–173. MIT Press, Cambridge (1985)
40. Li, H., Krishnamurthi, S., Fisler, K.: Verifying cross-cutting features as open systems. *SIGSOFT Softw. Eng. Notes* 27, 89–98 (2002)
41. Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.* 16(6), 1811–1841 (1994)
42. Madsen, O.L., Møller-Pedersen, B.: A unified approach to modeling and programming. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) *MODELS 2010. LNCS*, vol. 6394, pp. 1–15. Springer, Heidelberg (2010)
43. Manna, Z., Pnueli, A.: *The temporal logic of reactive and concurrent systems*. Springer, Heidelberg (1992)
44. McCarthy, J.: Circumscription—a form of non-monotonic reasoning. *Artificial Intelligence* 13, 27–39 (1980)

45. McCarthy, J., Hayes, P.J.: Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence* 4, 463–502 (1969)
46. McMullin, E.: Galilean Idealization. *Studies in the History and Philosophy of Science* 16, 247–273 (1985)
47. Miiikkulainen, R., Bednar, J.A., Choe, Y., Sirosh, J.: *Computational Maps in the Visual Cortex*. Springer, Heidelberg (2005)
48. Mitchell, J.C., Plotkin, G.D.: Abstract types have existential type. *ACM Trans. Program. Lang. Syst.* 10(3), 470–502 (1988)
49. Montague, R.: Universal grammar. In: *Formal Philosophy*, pp. 222–246 (1970)
50. Morris, J.H.: *Lambda-Calculus Models of Programming Languages*. PhD thesis, Massachusetts Institute of Technology (1968)
51. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Jul, E. (ed.) *ECOOP 1998*. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
52. Object Management Group (OMG). *Object management architecture guide*, ed 2.0. (1992)
53. O’Hearn, P.W., David, Pym, D.J.: The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 215–244 (1999)
54. O’Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: *Proc. Symp. Principles of Programming Languages (POPL)*, pp. 268–280. ACM, New York (2004)
55. Ossher, H., Tarr, P.: On the need for on-demand remodularization. In: *Position Paper for Aspects and Dimensions of Concern Workshop, ECOOP, Citeseer* (2000)
56. Ostermann, K.: Reasoning about aspects with common sense. In: *Proc. Int’l Conf. Aspect-Oriented Software Development (AOSD)*, pp. 48–59. ACM, New York (2008)
57. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: *Proc. Symp. Principles of Programming Languages (POPL)*, pp. 75–86. ACM, New York (2008)
58. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15(12), 1053–1058 (1972)
59. Parnas, D.L.: On a “buzzword”: Hierarchical structure. In: *Proceedings of IFIP Congress 1974*, pp. 336–339. North-Holland, Amsterdam (1974)
60. Parnas, D.L.: Use of abstract interfaces in the development of software for embedded computer systems. Technical report, *NRL Report No. 8047* (1977)
61. Parnas, D.L.: The secret history of information hiding. In: *Software Pioneers: Contributions to Software Engineering*, pp. 399–409. Springer, Heidelberg (2002)
62. Parnas, D.L.: Precise documentation: The key to better software. In: Nanz, S. (ed.) *The Future of Software Engineering*, pp. 125–148. Springer, Heidelberg (2011)
63. Parnas, D.L., Clements, P.C., Weiss, D.M.: The modular structure of complex systems. In: *Proc. Int’l Conf. Software Engineering (ICSE)*, pp. 408–417. IEEE Press, Los Alamitos (1984)
64. Popper, K.R.: *Conjectures and refutations: The growth of scientific knowledge*. Harper & Row, New York (1968)
65. Reiter, R.: On closed world data bases. In: *Logic and Data Bases*, pp. 55–76 (1977)
66. Reiter, R.: *A logic for default reasoning*, pp. 68–93. Morgan Kaufmann Publishers Inc., San Francisco (1987)
67. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: *IFIP Congress*, pp. 513–523 (1983)
68. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: *Proc. Symposium on Logic in Computer Science (LICS)*, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)

69. Rinard, M.C., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., Beebe, W.S.: Enhancing server availability and security through failure-oblivious computing. In: Proc. Symposium on Operating Systems Design & Implementation (OSDI), pp. 303–316 (2004)
70. Robillard, M., Murphy, G.C.: Concern graphs: Finding and describing concerns using structural program dependencies. In: Proc. Int’l Conf. Software Engineering (ICSE), pp. 406–416. ACM, New York (2002)
71. Ross, S.D.: Plato’s Theory of Ideas. Oxford University Press, Oxford (1951)
72. Roy, P.V., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge (2004)
73. Sandewall, E.: Nonmonotonic inference rules for multiple inheritance with exceptions. In: Expert systems, pp. 239–247. IEEE Computer Society Press, Los Alamitos (1990)
74. Smith, B.C.: Reflection and semantics in LISP. In: Proc. Symp. Principles of Programming Languages (POPL), pp. 23–35. ACM, New York (1984)
75. Stata, R., Guttag, J.V.: Modular reasoning in the presence of subclassing. In: Proc. Conf. Object-Oriented Programming, Systems, Languages & Applications (OOPSLA), pp. 200–214. ACM, New York (1995)
76. Steele, G.L.: Debunking the “expensive procedure call” myth or, procedure call implementations considered harmful or, LAMDBA: The ultimate GOTO. Technical report, Massachusetts Institute of Technology (1977)
77. Sternberg, R.: Cognitive Psychology. Thomson Wadsworth (2008)
78. Stoy, J.E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics. MIT Press, Cambridge (1977)
79. Tanenbaum, A.S., Herder, J.N., Bos, H.: Can we make operating systems reliable and secure? *Computer* 39, 44–51 (2006)
80. Tarr, P., Ossher, H., Harrison, W., Sutton Jr., S.M.: N degrees of separation: Multi-dimensional separation of concerns. In: Proc. Int’l Conf. Software Engineering (ICSE), pp. 107–119. IEEE Computer Society, Los Alamitos (1999)
81. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. *Inf. Comput.* 115(1), 38–94 (1994)

Worlds: Controlling the Scope of Side Effects

Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay

Viewpoints Research Institute
1209 Grand Central Ave.
Glendale, CA 91201
{alex,yoshiki,ted,alan}@vpri.org

Abstract. The state of an imperative program—e.g., the values stored in global and local variables, arrays, and objects’ instance variables—changes as its statements are executed. These changes, or side effects, are visible globally: when one part of the program modifies an object, every other part that holds a reference to the same object (either directly or indirectly) is also affected. This paper introduces *worlds*, a language construct that reifies the notion of program state and enables programmers to control the scope of side effects. We investigate this idea by extending both JavaScript and Squeak Smalltalk with support for worlds, provide examples of some of the interesting idioms this construct makes possible, and formalize the semantics of property/field lookup in the presence of worlds. We also describe an efficient implementation strategy (used in our Squeak-based prototype), and illustrate the practical benefits of worlds with two case studies.

1 Introduction

Solutions to many problems in computing start with incomplete information and must gather more while the solution is in progress. An important class of problems have to perform *speculations* and *experiments*, often in parallel, to discover how to proceed. These include classical non-deterministic problems such as certain kinds of parsing, search and reasoning, dealing with potential and actual error conditions, doing, undoing, and redoing in user interfaces, supporting multiple forked versions of files and other structures that may need to be both ramified and retracted, etc. The “need to undo” operates at all levels of scale in computing and goes beyond simple backtracking to being able to support multiple speculative world-lines.

Most of the ploys historically used to deal with “undoing” have been ad hoc and incomplete. For example, features such as `try/catch` enable some speculation, but only unwind the stack on failure; side effects are not undone automatically. Programmers have little choice but to rely on error-prone idioms such as the command design pattern [13]. This is analogous to the manual storage management mechanisms found in low-level languages (e.g., `malloc` and `free` in C). In contrast, garbage collection trades a little efficiency for enormous safety and convenience, and the *worlds* mechanism we present in this paper provides a similar service for all levels of “doing-and-undoing.”

Web surfing is a useful analogy for thinking about worlds: during a simple exploration of the web, you might just use the back button, but more complex explorations

(speculations) are more easily done with multiple tabs. All the changes you made during your explorations remain local to the tab that you used, and can be made “global” or not by your choice.

This is somewhat similar to *transactions*, which are another example of a general mechanism that can handle some of the “computing before committing” problems at hand here. But whereas the purpose of transactions is to provide a simple model for parallel programming, the goal of worlds is to provide a clean and flexible mechanism for *controlling the scope of side effects*. Unlike transactions, worlds are first-class values and are not tied to any particular control structure—a world can be stored in a variable to be revisited at a later time. This novel combination of design properties makes worlds more general (albeit more primitive) than transactions. For example, neither the module system shown in Section 3.3 nor the tree-undo feature presented in Section 6 could be implemented using transactions. Furthermore, we show in Section 8 that it is straightforward to implement transactions in a language that supports worlds.

The rest of paper is structured as follows. Section 2 introduces the notion of worlds and its instantiation in Worlds/JS. Section 3 illustrates some of the interesting idioms made possible by this construct. Section 4 details the semantics of property lookup in the presence of worlds. Section 5 describes the efficient implementation strategy used in our Squeak-based prototype. Sections 6 and 7 present two case studies, the first of which is used to benchmark the performance of our implementation. Section 8 compares worlds with related work, and Section 9 concludes.

2 Approach

The *world* is a new language construct that reifies the notion of program state. All computation takes place inside a world, which captures all of the side effects—changes to global and local variables, arrays, objects’ instance variables, etc.—that happen inside it.

Worlds are first-class values: they can be stored in variables, passed as arguments to functions, etc. They can even be garbage-collected just like any other object.

A new world can be “sprouted” from an existing world at will. The state of a *child world* is derived from the state of its *parent*, but the side effects that happen inside the child do not affect the parent. (This is analogous to the semantics of delegation in prototype-based languages with copy-on-write slots.) At any time, the side effects captured in the child world can be propagated to its parent via a *commit* operation.

2.1 Worlds/JS

A programming language that supports worlds must provide some way for programmers to:

- refer to the current world,
- sprout a new world from an existing world,
- commit a world’s changes to its parent world, and
- execute code in a particular world.

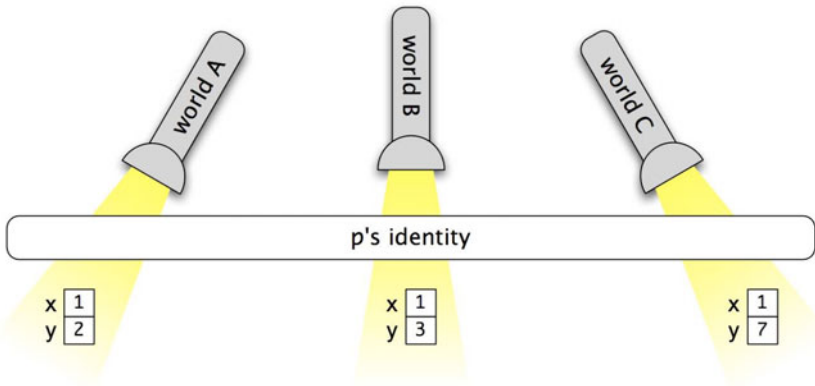


Fig. 1. Projections/views of the same object in three different worlds

We now describe the way in which these operations are supported in Worlds/JS, an extension of JavaScript [9] we have prototyped in order to explore with the ideas discussed in this paper. (Worlds/JS is available at http://www.tinlizzie.org/ometa-js/#Worlds_Paper. No installation is necessary; you can experiment with the language directly in your web browser.)

Worlds/JS extends JavaScript with the following new syntax:

- `thisWorld` — is an expression whose value is the world in which it is evaluated (i.e., the “current world”), and
- `in <expr> <block>` — is a statement that executes `<block>` inside the world obtained from evaluating `<expr>`.

All worlds delegate to the world prototype, whose `sprout` and `commit` methods can be used to create a new world that is a child of the receiver, and propagate the side effects captured in the receiver to its parent, respectively.

In the following example, we modify the `y` property of the same instance of `Point` in two different ways, each in its own world, and then commit one of them to the original world. This serves the dual purpose of illustrating the syntax of Worlds/JS and the semantics of the *sprout* and *commit* operations.

```
A = thisWorld;
p = new Point(1, 2);

B = A.sprout();
in B { p.y = 3; }

C = A.sprout();
in C { p.y = 7; }

C.commit();
```

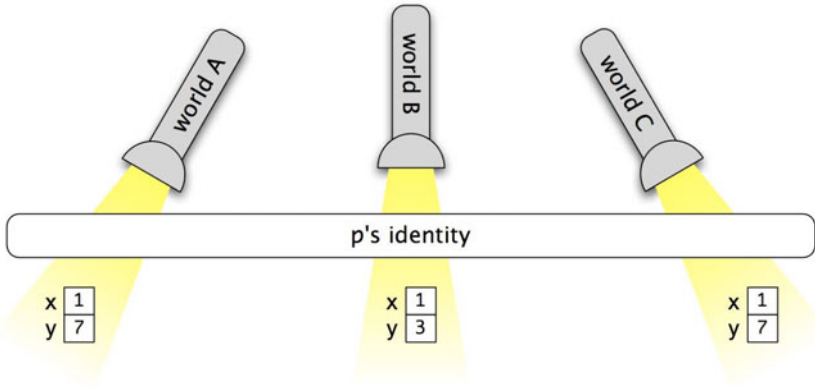


Fig. 2. The state of the “universe” shown in Figure 1 after a *commit* from world C

Figures 1 and 2 show the state of the point in each world, before and after the *commit* operation, respectively. Note that *p*’s identity is “universal,” and each world associates it with *p*’s state in that world.

2.2 Safety Properties

Programming with worlds should not be error-prone or dangerous. In particular, if w_{child} is a world that was sprouted from w_{parent} :

- Changes in w_{parent} —whether *explicit* (caused by assignments in w_{parent} itself) or *implicit* (caused by a *commit* from one of w_{child} ’s siblings)—should never make variables appear to change spontaneously in w_{child} . We call this the “no surprises” property.
- Similarly, a *commit* from w_{child} should never leave w_{parent} in an inconsistent state, e.g., because the changes being committed are incompatible with changes made in w_{parent} after w_{child} was sprouted. We call this property “consistency.”

In this section, we explain how the semantics of worlds ensures these properties.

Preventing “Surprises”. Once a variable (or slot, memory location, etc.) has been read or modified in a world w , subsequent changes to that variable in w ’s parent world are not visible in w . This ensures that variables do not appear to change spontaneously in child worlds.

For example, Figure 2 shows that the effects of the *commit* from world C ($p.y \leftarrow 7$) are not visible in world B (because it has also modified $p.y$). However, if yet another world that is sprouted from A changes the value of $p.x$ and then *commits*, as shown below,

```
D = A.sprout();
in D {
  p.x = 5;
}
D.commit();
```

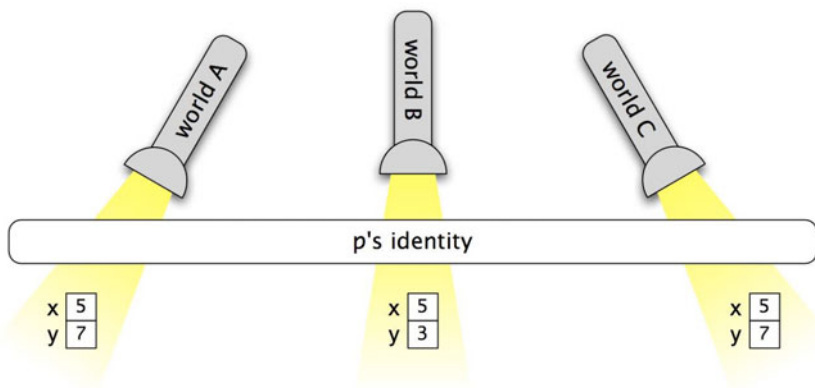


Fig. 3. The state of the “universe” shown in Figure 2 after a *commit* from world D (not pictured)

the new value of $p.x$ becomes visible not only in A, because it is D’s parent, but also in B and C, since neither of them has read or modified $p.x$. (See Figure 3.) This is safe because up to this point:

- neither B nor C has read the old value of $p.x$, so they will not be able to tell that it has changed, and
- whatever writes have already been done in B and C (assuming the program is correct) are guaranteed not to depend on the value of $p.x$ —otherwise $p.x$ would have been read in those worlds.

Preserving Consistency. Note that the “no surprises” property does not prevent the state of a parent world from changing as a result of *commits* from its children—after all, the sole purpose of the commit operation is to change the state of the parent world. But not all such changes are safe: certain kinds of changes could leave the parent world in an inconsistent state. This is why the commit operation of worlds, like its counterpart in transaction-processing systems, is guarded by a *serializability check*.

A *commit* from w_{child} to w_{parent} is only allowed to happen if, at *commit-time*, all of the variables (or slots, memory locations, etc.) that were read in w_{child} have the same values in w_{parent} as they did when they were first read by w_{child} . If this is not the case, some of the assignments that were made in w_{child} may have been based on values that are now out of date. A *commit* that fails the serializability check is aborted, leaving both child and parent worlds unchanged, and throws a `CommitFailed` exception.

Section 5 describes the implementation of the commit operation, including the serializability check described here.

3 Worlds by Example

The following examples illustrate some of the applications of worlds. Other obvious applications (not discussed here) include sand-boxing and heuristic search.

3.1 Better Support for Exceptions

In languages that support exception-handling mechanisms (e.g., the `try/catch` statement), a piece of code is said to be *exception-safe* if it guarantees not to leave the program in an inconsistent state when an exception is thrown. Writing exception-safe code is a tall order, as we illustrate with the following example:

```
try {
    for (var idx = 0; idx < xs.length; idx++)
        xs[idx].update();
} catch (e) {
    // ...
}
```

Our intent is to update every element of `xs`, an array. The problem is that if one of the calls to `update` throws an exception, some (but not all) of `xs`' elements will have been updated. So in the `catch` block, the program should restore `xs` to its previous consistent state, in which none of its elements was updated.

One way to do this might be to make a copy of every element of the array before entering the loop, and in the `catch` block, restore the successfully-updated elements to their previous state. In general, however, this is not sufficient since `update` may also have modified global variables and other objects on the heap. Writing truly exception-safe code is difficult and error-prone.

Versioning exceptions [23] offer a solution to this problem by giving `try/catch` statements a transaction-like semantics: if an exception is thrown, all of the side effects resulting from the incomplete execution of the `try` block are automatically rolled back before the `catch` block is executed. In a programming language that supports worlds and a traditional (non-versioning) `try/catch` statement, the semantics of versioning exceptions can be implemented as a design pattern. We illustrate this pattern with a rewrite of the previous example:

```
try {
    in thisWorld.sprout() {
        for (var idx = 0; idx < xs.length; idx++)
            xs[idx].update();
        thisWorld.commit();
    }
} catch (e) {
    // no clean-up required!
}
```

Note that the statements of the original `try` block are now evaluated in a new world that will capture their side effects. Note also that inside the `in` statement, the pseudo-variable `thisWorld` refers to this new world, and not its parent world. Therefore, if the loop terminates normally (i.e., without throwing an exception), the statement `thisWorld.commit();` will propagate the side effects to the parent world. On the other hand, if an exception is thrown, control will pass to the `catch` block before the `commit`

operation is executed, and thus the side effects will be discarded. (In fact, the new world will eventually be garbage-collected, since it is not referenced by any variables.)

3.2 Undo for Applications

We can think of the “automatic clean-up” supported by versioning exceptions as a kind of one-level *undo*. In the last example, we implemented this by capturing the side effects of the `try` block—the operation we may need to undo—in a new world. The same idea can be used as the basis of a framework that makes it easy for programmers to implement applications that support multi-level undo.

Applications built using this framework are objects that support two operations: `perform` and `undo`. Clients use the `perform` operation to issue commands to the application, and the `undo` operation to restore the application to its previous state (i.e., the state it was in before the last command was performed). The example below illustrates how a client might interact with a counter application that supports the commands `inc`, `dec`, and `getCount`, for incrementing, decrementing, and retrieving the counter’s value, respectively. (The counter’s value is initially zero.)

```
counter.perform('inc');
counter.perform('inc');
counter.perform('dec');
counter.undo(); // undo 'dec'
print(counter.perform('getCount')); // outputs '2'
```

The interesting thing about our framework is that it allows programmers to implement applications that support multi-level undo *for free*, i.e., without having to use error-prone idioms such as the *command* design pattern [13]. The implementation of the counter application—or rather, a factory of counters—is shown below:

```
makeCounter = function() {
  var app      = new Application();
  var count    = 0;
  app.inc      = function() { count++; };
  app.dec      = function() { count--; };
  app.getCount = function() { return count; };
  return app;
};
```

Note that the counter application is an instance of the `Application` class. `Application` is our framework; in other words, it is where all of the undo functionality is implemented. Its source code is shown in Figure 4.

The state of the application is always accessed in a world that “belongs” to the application. When the application is instantiated, it has only one world. Each time a client issues a command to the application via its `perform` operation, the method that corresponds to that command (the one with the same name as the command) is invoked in a new world. This new world is sprouted from the world that holds the previous version of the application’s state (i.e., the one in which the last command was executed).

```

Application = function() { };
Application.prototype = {
  worlds: [thisWorld],
  perform: function(command) {
    var w = this.worlds.last().sprout();
    this.worlds.push(w);
    in w { return this[command](); }
  },
  undo: function() {
    if (this.worlds.length > 0)
      this.worlds.pop();
  },
  flattenHistory: function() {
    while (this.worlds.length > 1) {
      var w = this.worlds.pop();
      w.commit();
    }
  }
};

```

Fig. 4. A framework for building applications that support multi-level undo

The undo operation simply discards the world in which the last command was executed, effectively returning the application to its previous state. Lastly, the (optional) `flattenHistory` operation coalesces the state of the application into a single world, which prevents clients from undoing past the current state of the application.

Note that the application’s public interface (the `perform` and `undo` methods) essentially models the way in which web browsers interact with online applications, so this technique could be used in a web application framework like *Seaside* [8].

Our Worlds/Squeak image includes a text editor implementation that supports multi-level undo using the idiom described in this section. It is available for download at <http://www.tinlizzie.org/~awarth/worlds>.

3.3 Extension Methods in JavaScript

In JavaScript, functions and methods are “declared” by assigning into properties. For example,

```

Number.prototype.fact = function() {
  if (this == 0)
    return 1;
  else
    return this * (this - 1).fact();
};

```

adds the factorial method to the `Number` prototype. Similarly,

```

inc = function(x) { return x + 1 };

```

declares a function called `inc`. (The left-hand side of the assignment above is actually shorthand for `window.inc`, where `window` is bound to JavaScript’s *global* object.)

JavaScript does not support modules, which makes it difficult, sometimes even impossible for programmers to control the scope of declarations. But JavaScript's declarations are really side effects, and worlds enable programmers to control the scope of side effects. We believe that worlds could serve as the basis of a powerful module system for JavaScript, and have already begun experimenting with this idea.

Take extension methods, for example. In dynamic languages such as JavaScript, Smalltalk, and Ruby, it is common for programmers to extend existing objects/classes (e.g., the `Number` prototype in JavaScript) with new methods that support the needs of their particular application. This practice is informally known as *monkey-patching* [4]. Monkey-patching is generally frowned upon because, in addition to polluting the interfaces of the objects involved, it makes programs vulnerable to name clashes that are impossible to anticipate. Certain module systems, including those of MultiJava [6] and eJava [33], eliminate these problems by allowing programmers to declare *lexically-scoped* extension methods. These must be explicitly imported by the parts of an application that wish to use them, and are invisible to the rest of the application.

The following example shows that worlds can be used to support this form of modularity:

```
ourModule = thisWorld.sprout();
in ourModule {
  Number.prototype.fact = function() { ... };
}
```

The factorial method defined above can only be used inside `ourModule`, e.g.,

```
in ourModule {
  print((5).fact());
}
```

and therefore does not interfere with other parts of the program.

This idiom can also be used to support *local rebinding*, a feature found in some module systems [3|2|7] that enables programmers to locally replace the definitions of existing methods. As an example, we can change the behavior of `Number`'s `toString` method only when used inside `ourModule`:

```
in ourModule {
  numberToEnglish = function(n) { ... };
  Number.prototype.toString = function() {
    return numberToEnglish(this);
  };
}
```

and now the output generated by

```
arr = [1, 2, 3];
print(arr.toString());
in ourModule {
  print(arr.toString());
}
```


is

```
[1, 2, 3]
[one, two, three]
```

A more detailed discussion of how worlds can be used to implement a module system for JavaScript, including a variation of the idiom described above that supports side-effectful extension methods, can be found in the first author’s Ph.D. dissertation [31].

4 Property Lookup Semantics

This section formally describes the semantics of property lookup in Worlds/JS, which is a natural generalization of property lookup in JavaScript. We do not formalize the semantics of field lookup in Worlds/Squeak, since it is just a special case of the former in which all prototype chains have length 1 (i.e., there is no delegation).

4.1 Property Lookup in JavaScript

JavaScript’s object model is based on single delegation, which means that every object inherits (and may also override) the properties of its “parent” object. The only exception to this rule is `Object.prototype` (the ancestor of all objects), which is the root of JavaScript’s delegation hierarchy and therefore does not delegate to any other object.

The semantics of property lookup in JavaScript can be formalized using the following two primitive operations:

- (i) *getOwnProperty*(x, p), which looks up property p in object x *without looking up the delegation chain*. More specifically, the value of *getOwnProperty*(x, p) is
 - v , if x has a property p that is not inherited from another object, and whose value is v , and
 - the special value **none**, otherwise;
- (ii) *parent*(x), which evaluates to
 - y , the object to which x delegates, or
 - the special value **none**, if x does not delegate to any other object.

and the following set of inference rules:

$$\frac{\text{getOwnProperty}(x, p) = v \quad v \neq \mathbf{none}}{\text{lookup}(x, p) = v} \quad (\text{JS-LOOKUP-OWN})$$

$$\frac{\text{getOwnProperty}(x, p) = \mathbf{none} \quad \text{parent}(x) = \mathbf{none}}{\text{lookup}(x, p) = \mathbf{none}} \quad (\text{JS-LOOKUP-ROOT})$$

$$\frac{\text{getOwnProperty}(x, p) = \mathbf{none} \quad \text{parent}(x) = y \quad y \neq \mathbf{none} \quad \text{lookup}(y, p) = v}{\text{lookup}(x, p) = v} \quad (\text{JS-LOOKUP-CHILD})$$

4.2 Property Lookup in Worlds/JS

In Worlds/JS, property lookup is always done in the context of a world. And since it may be that an object x has a property p in some world w but not in another, the primitive operation $getOwnProperty(x, p)$ must be replaced by a new primitive operation, $getOwnPropertyInWorld(x, p, w)$.

Another primitive operation we will need in order to formalize the semantics of property lookup in Worlds/JS is $parentWorld(w)$, which yields w 's parent, or the special value **none**, if w is the top-level world.

Using these two new primitive operations, we can define a new operation, $getOwnProperty(x, p, w)$, which yields the value of x 's p property in world w , or (if $x.p$ is not defined in w) in w 's closest ancestor:

$$\frac{getOwnPropertyInWorld(x, p, w) = v}{v \neq \mathbf{none}} \quad (\text{WJS-GETOWNPROPERTY-OWN})$$

$$\frac{getOwnPropertyInWorld(x, p, w) = \mathbf{none} \quad parentWorld(w) = \mathbf{none}}{getOwnProperty(x, p, w) = \mathbf{none}} \quad (\text{WJS-GETOWNPROPERTY-ROOT})$$

$$\frac{getOwnPropertyInWorld(x, p, w_1) = \mathbf{none} \quad parentWorld(w_1) = w_2 \quad w_2 \neq \mathbf{none} \quad getOwnProperty(x, p, w_2) = v}{getOwnProperty(x, p, w_1) = v} \quad (\text{WJS-GETOWNPROPERTY-CHILD})$$

And finally, using the worlds-friendly variant of $getOwnProperty$ defined above, the inference rules that formalize the semantics of lookup in Worlds/JS can be written as follows:

$$\frac{getOwnProperty(x, p, w) = v}{v \neq \mathbf{none}} \quad (\text{WJS-LOOKUP-OWN})$$

$$\frac{getOwnProperty(x, p, w) = \mathbf{none} \quad parent(x) = \mathbf{none}}{lookup(x, p, w) = \mathbf{none}} \quad (\text{WJS-LOOKUP-ROOT})$$

$$\frac{getOwnProperty(x, p, w) = \mathbf{none} \quad parent(x) = y \quad y \neq \mathbf{none} \quad lookup(y, p, w) = v}{lookup(x, p, w) = v} \quad (\text{WJS-LOOKUP-CHILD})$$

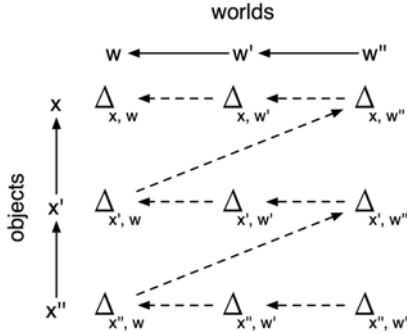


Fig. 5. The property lookup order used when evaluating $x''.p$ in world w'' (the notation $\Delta_{x,w}$ represents the properties of x that were modified in w)

Note that these rules closely mirror those that describe the semantics of lookup in JavaScript—the only difference is that *getOwnProperty* and *lookup* now each take a world as an additional argument.

Figure 5 illustrates the property lookup order that results from the algorithm described above. The solid vertical lines in the diagram indicate *delegates-to* relationships (e.g., object x' delegates to x), whereas the solid horizontal lines indicate *is-child-of* relationships (e.g., world w' is a child of w). Note that the chain of worlds gets precedence over the object delegation chain; in other words, any relevant “version” of an object may override the properties of the object to which it delegates. This lookup order preserves JavaScript’s copy-on-write delegation semantics, i.e., if a delegates to b , and then we assign into a ’s p property, subsequent changes to b ’s p property will not affect a . So no matter what world a statement is executed in—whether it is the top-level world, or a world that sprouted from another world—it will behave in exactly the same way as it would in “vanilla” JavaScript.

5 Implementation

Our Worlds/JS prototype works by translating Worlds/JS programs into JavaScript code that can be executed directly in a standard web browser, and it is useful for doing “quick and dirty” experiments. However, JavaScript’s lack of support for weak references required a costly work-around that makes Worlds/JS unsuitable for larger experiments.

In this section, we describe the implementation strategy used in our more performant prototype of worlds, which is based on Squeak Smalltalk [16]. (The Worlds/Squeak image, which also includes the case study discussed in Section 6, is available for download at <http://www.tinlizzie.org/~awarth/worlds/>.)

5.1 Data Structures

The core of our implementation consists of two classes: `WObject`, which is the superclass of all “world-friendly” objects (i.e., objects that exhibit the correct behavior when

viewed/modified inside worlds), and `WWorld`, which represents worlds. The methods of `WObject` and its subclasses are automatically instrumented in order to indirect all instance variable accesses (reads and writes) through the world in which they are being evaluated.

A `WWorld` w contains a hash table that, similar to a *transaction log*, associates `WObjects` with

- a *reads* object, which holds the “old” values of each slot of the `WObject` when it was first read in w , or the special `Don'tKnow` value for each slot that was never read in w , and
- a *writes* object, which holds the most recent values of each slot of the `WObject`, or the special `Don'tKnow` value for slots that were never written to in w .

The keys of this hash table are referenced weakly to ensure that the *reads* and *writes* objects associated with a `WObject` that is no longer referenced by the program will be garbage collected. Also, *reads* and *writes* objects are instantiated lazily, so (for example) an object that has been read but not written to in a world will have a *reads* object, but not a *writes* object, in that world.

5.2 The Slot Update Operation: $(x_i \leftarrow v)_w$

To store the value v in x 's i^{th} slot in world w ,

1. If w does not already have a *writes* object for x , create one.
2. Write v into the i^{th} slot of the *writes* object.

5.3 The Slot Lookup Operation: $(x_i)_w$

To retrieve the value stored in x 's i^{th} slot in world w ,

1. Let $w_{\text{current}} = w$ and $\text{ans} = \text{undefined}$.
2. If w_{current} has a *writes* object for x and the value stored in the i^{th} slot of the *writes* object is not `Don'tKnow`, set ans to that value and go to step 5.
3. If w_{current} has a *reads* object for x and the value stored in the i^{th} slot of the *reads* object is not `Don'tKnow`, set ans to that value and go to step 5. (This step ensures the “no surprises” property, i.e., that a slot value does not appear to change spontaneously in w when it is updated in one of w 's ancestors.)
4. Otherwise, set w_{current} to w_{current} 's parent, and go to step 2.
5. If w does not already have a *reads* object for x , create one.
6. If the value stored in the i^{th} slot of the *reads* object is `Don'tKnow`, write ans into that slot.
7. Return ans .

Note that the slots of a new `WObject` are always initialized with `nil`s in the top-level world. This mirrors the semantics of object instantiation in Smalltalk and ensures that lookup always terminates.

(We initially implemented the slot lookup operation in Smalltalk, but later re-implemented it as a primitive, which resulted in a significant performance improvement. See Section [6.3](#) for details.)

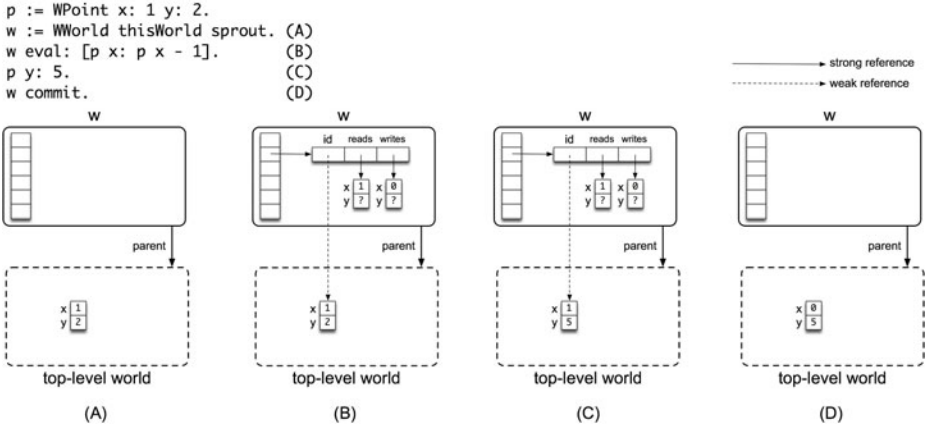


Fig. 6. A successful commit

5.4 Reads and Writes in the Top-Level World

The top-level world has no need for *reads* objects, since it has no parent world to commit to. This gave us an idea: if we made every *WObject* be its own *writes* object in the top-level world, there would be no need for a hash table. This optimization minimizes the overhead of using *WObjects* in the top-level world—after all, the slot update and lookup operations can manipulate the *WObject* directly, without the additional cost of a hash table lookup.

5.5 The Commit Operation

To commit the side effects captured in world w_{child} to its parent, w_{parent} ,

1. **Serializability check:** for all $x_i = v$ in each of w_{child} 's *reads* objects, make sure that either $v = \text{Don'tKnow}$ or the current value of x_i in w_{parent} is equal to v . (Otherwise, throw a `CommitFailed` exception.)
2. Propagate all of the information in w_{child} 's *writes* objects to w_{parent} 's *writes* objects, overriding the values of any slots that have already been assigned into in w_{parent} .
3. Propagate all of the information in w_{child} 's *reads* objects to w_{parent} 's *reads* objects, except for the slots that have already been read from in w_{parent} . (This step ensures that the serializability check associated with a later *commit* from w_{parent} will protect the consistency of its own parent.)
4. Clear w_{child} 's hash table.

Note that in step 2, new *writes* objects must be created for any objects that were written to in w_{child} , but not in w_{parent} . Similarly, in step 3, new *reads* objects must be created for any objects that were read from in w_{child} , but not in w_{parent} (unless w_{parent} is the top-level world, which, as discussed in Section 5.4, has no need for *reads* objects).

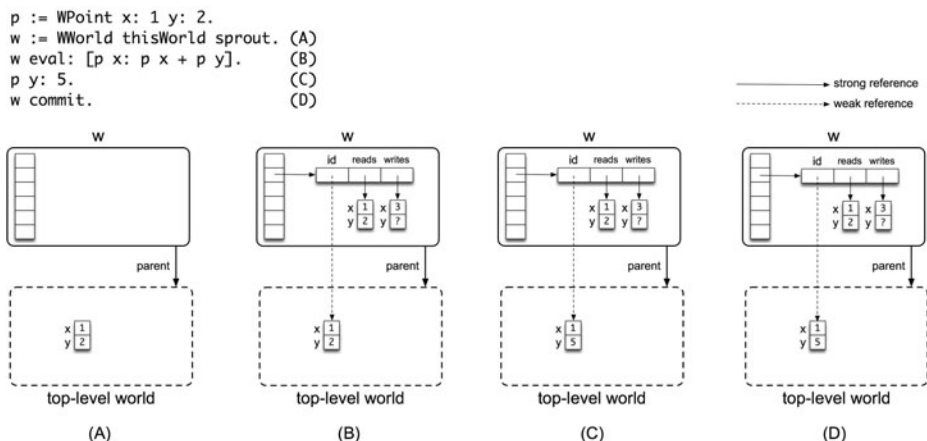


Fig. 7. A failed commit

5.6 Pulling It all Together

The Worlds/Squeak code and diagrams in Figures 6 and 7 show these data structures and operations in action.

In Figure 6 (A), we see a point p ¹ and a world w that was sprouted from the top-level world. In (B), we see what happens when we decrement p 's x field in w . (`WWorld>>eval:` is equivalent to the `in`-statement in Worlds/JS.) Note that this step results in the creation of a hash table entry for p . (Note also that the key used to index the hash table is p itself—the identity of an object never changes, no matter what world it's in.) The old (1) and new (0) values of x are stored in p 's *reads* and *writes* objects, respectively; the question marks denote the *Don'tKnow* value. In (C), p 's y field is updated in the top-level world with the value 5. Lastly, (D) shows the effects of a commit on w . The old values stored in the *reads* objects in the child world (in this case there is only one, $p.x = 1$) are compared against the corresponding (current) values in the parent world. Since these are equal, the changes that were stored in the child world ($p.x \leftarrow 0$) are propagated to the parent world. Note that a successful *commit* does not cause the child world to be discarded; it simply clears the information stored in that world so that it can be used again.

Figure 7 illustrates a slightly different scenario. In (B), both x and y are read in order to compute the new value of x and as a result, both values are recorded in the *reads* object. In (C), y is updated in the top-level world. In (D), w tries to commit but fails the serializability check because the value of y that was recorded in the *reads* object is different from the current value of y in the top-level world. This results in a `CommitFailed` exception, and the commit operation is aborted. Note that (C) and (D) are identical—a failed *commit* leaves both the parent and child worlds unchanged. This enables the programmer to examine (and also extract) potentially useful values in the child world

¹ p is an instance of `WPoint`, which is a subclass of `WObject`.

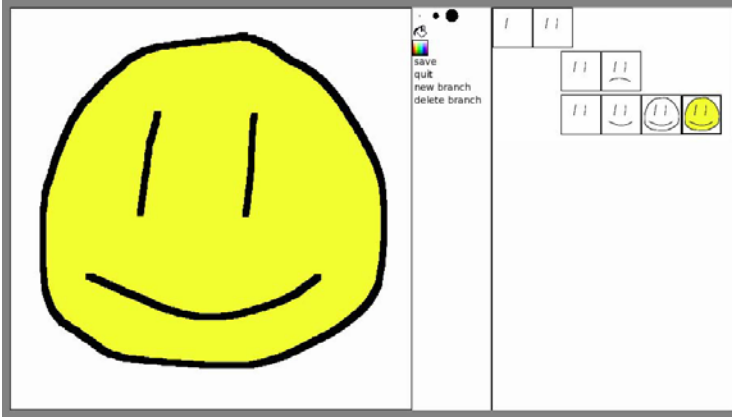


Fig. 8. Our bitmap editor, implemented in Worlds/Squeak, supports tree-undo

using its `#eval:` method. The programmer may also start anew by sprouting a new child world.

6 Case Study #1: A Bitmap Editor

To gauge the utility of worlds in practical applications, we have implemented a bitmap editor that supports a sophisticated tree-undo mechanism. This is an interesting challenge for worlds because it involves large bitmap objects ($\sim 1\text{M}$ slots per bitmap) that must not only be manipulated interactively and efficiently, but also passed to Squeak’s graphics primitives, which do not support worlds.

Applications that support (traditional) linear undo allow users to undo/redo a series of actions. *Tree-undo* is a more general model for undo that enables users to create new “branches” in the edit history and move from freely from branch to branch, which in turn makes it convenient for different choices/possibilities to be explored concurrently.

Figure 8 shows the Graphical User Interface (GUI) of our bitmap editor. To the right of the large painting area there are several tool-selecting buttons: three different brushes, a bucket fill tool, and a color picker. Below these buttons, in addition to the self-explanatory “save” and “quit,” there are buttons for managing the undo tree: “new branch” creates a new branch that stems from the current undo record, and “delete branch” discards the current branch. Last but not least, to the right of these buttons is a view of the undo tree in which each thumbnail represents an undo record (the one with the thick border is the current undo record), and each row of thumbnails represents a sequence of states in the same branch.

While our bitmap editor (which comprises approximately 300 lines of Lesser-Phic2/Squeak code) is not particularly feature-rich, it is interesting enough to serve as a benchmark for measuring the performance of Worlds/Squeak.

6.1 The Implementation of Tree Undo

As illustrated in Section 3.2, the semantics of worlds provides a good basis for implementing the undo mechanism: all we have to do is create a chain of worlds in which to capture changes to the state of the application. The undo operation, then, involves merely switching the world from which we view the state of the application. This idiom gives the application designer complete control over what parts of the application’s state should be undoable (i.e., what parts are always modified in and viewed from a world in the undo chain), and the granularity of the undo operation (i.e., how often new worlds, or “undo records,” are created).

In our bitmap editor, the only piece of application state we chose to make undoable was the bitmap itself. So whenever the user performs an action that modifies the bitmap, e.g., a pen stroke, we sprout a new world from the current world, and make the appropriate changes to the bitmap in that new world. After drawing three strokes for example, we are left with a chain of four worlds (the first represents the state of the bitmap before the first stroke).

Extending this idiom to support tree-undo was straightforward. To make a new branch, we simply sprout a new world from the current world, and to delete a branch, we remove all references to the root of that branch (as a result, its undo records and their associated worlds will eventually be garbage-collected). Our program must maintain undo records, which are data structures that represent the branches of the undo tree, because a world does not retain a list of its children (after all, this would prevent worlds from ever being garbage-collected, since every world is a descendant of the top-level world).

6.2 Bitmap Representation

We wanted our editor to feel “smooth.” A user should, for example, be able to edit a 500×500 -pixel bitmap while the screen is updated 30 times per second. This meant that our application had to sustain 7.5M slot lookup operations per second just to display the bitmap. Our first (naive) implementation of Worlds/Squeak, in which `WObject`’s lookup operation was implemented entirely in Smalltalk, did not meet these requirements: while drawing a line on a 2.4 GHz Core Duo computer, we observed a refresh rate of about 1 fps—far from usable in any reasonable standard.

To increase the performance of our application, we created a variable-length, non-pointer subclass of `WObject` called `WBitmap`. We made `WBitmaps` structurally identical to Squeak’s `Bitmaps` so that the `BitBlt` primitives could be used to draw on `WBitmaps`.² We implemented “write-only” features like line drawing, for example, by passing the `WBitmap`’s “delta” to a `BitBlt` primitive that efficiently mutates it as desired. (By modifying the delta, we ensure that the changes are only visible in the appropriate world.)

We also realized that `WObject`’s slot lookup method could be implemented as a Squeak primitive; in fact, we took this idea one step further and implemented a primitive that reads *all of the slots* of a `WObject`, and returns a “flattened snapshot” of that object, as shown in Figure 9. Note that, because all of its state is stored in a single Squeak object (i.e., in a contiguous block of memory), a flattened `WObject` can be used

² Some primitives turned out to have strict type checks that prevented us from using them.

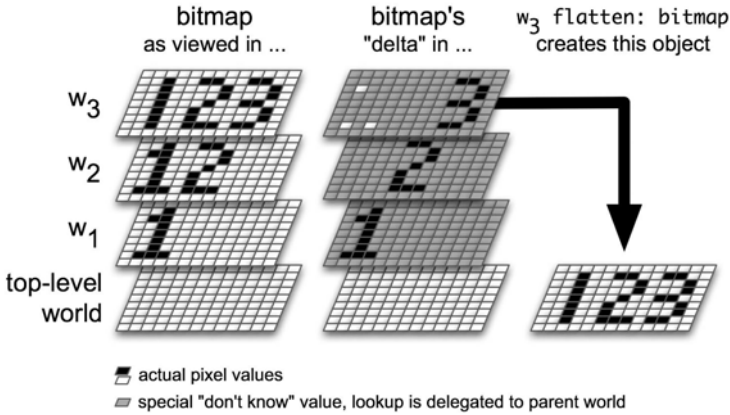


Fig. 9. The flatten operation

to interface with parts of the system that are not “worlds-aware,” including Squeak’s graphics system. To flood-fill an area of a `WBitmap`, for instance, we pass the bitmap’s delta *and* a flattened snapshot to a primitive; the flattened object is used for reading values out of the bitmap, and the delta is used as a target for the writes.

The downside of representing `WBitmaps` as arrays of “immediate” (i.e., non-pointer) values is that it makes it impossible for their slots to reference the special `Don’tKnow` value that causes lookup to be delegated to the parent world—after all, every 4-byte value is a valid (ARGB) pixel value. We got around this problem by using the value “0x00000001” (the darkest, completely transparent blue) as `WBitmap`’s equivalent of `WObject`’s `Don’tKnow` value. A more general work-around—which would be necessary for applications that require access to all possible 4-byte values—would be to use an additional bit array (one bit per slot in the object) whose values indicate whether or not a slot in a “delta” or “orig” object contains a valid value.

6.3 Benchmarking the Bitmap Editor

After we introduced the flatten primitive, our bitmap editor became **really responsive**: while editing a 500×500 -pixel bitmap on the same 2.4 GHz Core Duo computer, the frame rate only drops below 30 once the length of the delegation chain—i.e., the distance between the current undo record and the root of the undo tree—reaches about 50. (At this point, a casual user may start to notice a slowdown.)

In order to get a better idea of the performance characteristics of our bitmap editor (and of our Squeak-based implementation of worlds), we conducted the following experiment. We started with a 20,000-pixel bitmap, created a chain of worlds, and measured the time it takes to read all of the slots (pixel values) of the bitmap from the world at the end of the chain. Looking up a slot’s value requires a traversal of the world chain until a valid (non-`Don’tKnow`) value is found, so the “load factor,” i.e., the number of value-holding slots at each level, dictates the typical length a lookup operation, which in turn determines the performance of the application.

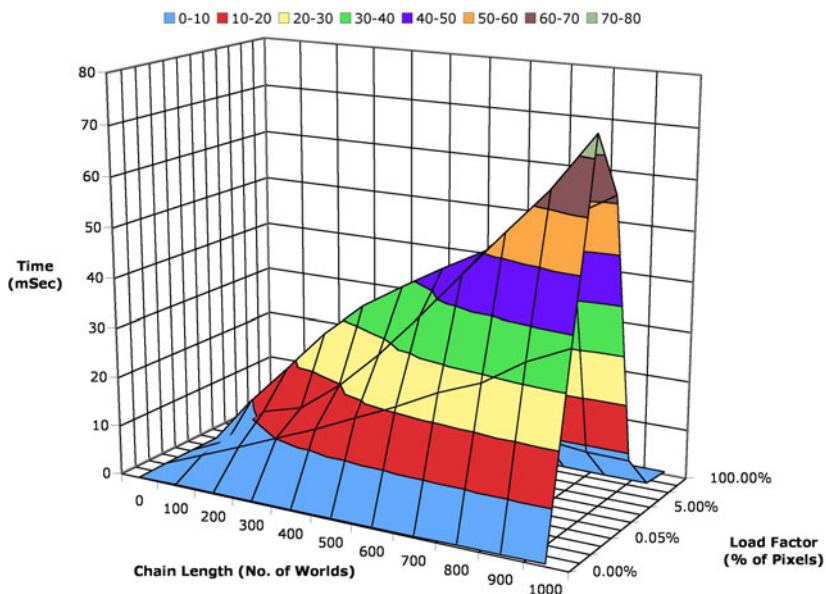


Fig. 10. time to flatten a 20,000-slot bitmap

Figure 10 shows the results of this experiment. When the load factor is close to zero, the lookup primitive must nearly always scan all of the worlds in the chain, and as a result, time grows linearly with the number of worlds in the chain. When the load factor is high, e.g., when 50% of the slots are filled randomly at each level, the lookup primitive only has to inspect a few worlds—two, on average—so the length of the chain has no measurable effect on the performance of our application.

The “lookup” and “flatten” primitives provide significant performance improvements. Without them, reading every slot of a 500×500 -pixel bitmap through a chain of 1000 worlds takes 27.17 seconds and 137 milliseconds with a load factor of 0.005% and 50%, respectively. Using the “lookup” primitive to access each slot reduces these times to 0.36 seconds and 10 milliseconds, respectively. Using the “flatten” primitive further reduces these times to 0.038 seconds and less than 1 millisecond, respectively. On average, the use of these primitives improved the performance of our application by two orders of magnitude.

7 Case Study #2: OMeta + Worlds

Consider the semantics of the ordered choice operator ($|$) in OMeta [32], or if you prefer, in Parsing Expression Grammars and Packrat Parsers [11][12]. If a match fails while the first operand is being evaluated, the parser, or more generally, the *matcher* has to backtrack to the appropriate position on the input stream before trying the second operand. We can think of this backtracking as a limited kind of undo that is only

```

OMeta._or = function() {
  var origInput = this.input;
  for (var idx = 0;
       idx < arguments.length;
       idx++) {

    try {
      this.input = origInput;
      return arguments[idx]();
    }
    catch (f) {

      if (f != fail)
        throw f;
    }

    throw fail;
  };
}

```

(A)

```

OMeta._or = function() {
  for (var idx = 0;
       idx < arguments.length;
       idx++) {
    var ok = true;
    in thisWorld.sprout() {
      try {
        return arguments[idx]();
      }
      catch (f) {
        ok = false;
        if (f != fail)
          throw f;
      }
      finally {
        if (ok)
          thisWorld.commit();
      }
    }
  }
  throw fail;
};

```

(B)

Fig. 11. Two different implementations of OMeta’s ordered choice operator

concerned with changes to the matcher’s position on the input stream. Other kinds of side effects that can be performed by semantic actions—e.g., destructive updates such as assigning into one of the fields of the matcher object or a global variable—are not undone automatically, which means that the programmer must be specially careful when writing rules with side effects.

To show that worlds can greatly simplify the management of state in backtracking programming languages like OMeta and Prolog, we have implemented a variant of OMeta/JS [30] in which the choice operator automatically discards the side effects of failed alternatives, and similarly, the repetition operator (*) automatically discards the side effects of the last (unsuccessful) iteration. This was modification straightforward: since Worlds/JS is a superset of JavaScript—the language in which OMeta/JS was implemented—all we had to do was redefine the methods that implement the semantics of these two operators.

Figures 11 (A) and (B) show the original and modified implementations of the ordered choice operator, respectively. Note that the modified implementation sprouts a new world in which to evaluate each alternative, so that the side effects of failed alternatives can be easily discarded. These side effects include the changes to the matcher’s input stream position, and therefore the code that implemented backtracking in the

original version (`this.input = origInput`) is no longer required. Finally, the side effects of the first successful alternative are committed to the parent world in the `finally` block.

Similarly, the alternative implementation of the repetition operator (omitted for brevity) sprouts a new world in which to try each iteration, so that the effects of the last (unsuccessful) iteration can be discarded.

And thus, with very little additional complexity, worlds can be used to make the use of side effects safer and easier to reason about in the presence of backtracking.

8 Related Work

In languages that support Software Transactional Memory (STM) [27,114], every transaction that is being executed at a given time has access to its own view of the program store that can be modified in isolation, without affecting other transactions. Therefore, like worlds, STM enables multiple versions of the store to co-exist. But whereas the purpose of transactions is to provide a simple model for parallel programming, the goal of worlds is to provide a clean and flexible mechanism for *controlling the scope of side effects*. Unlike transactions, worlds are first-class values and are not tied to any particular control structure—a world can be stored in a variable to be revisited at a later time. This novel combination of design properties makes worlds more general (albeit more primitive) than transactions. For example, neither the module system shown in Section 3.3 nor the tree-undo feature presented in Section 6 could be implemented using transactions. Furthermore, in a language that supports threads, it would be straightforward to implement the semantics of STM using worlds:

```
executeInNewThread {
  in thisWorld.sprout() {
    ... // statements to run in the transaction
    thisWorld.commit();
  }
}
```

(The code above assumes that commits are executed atomically.)

Burckhardt et al.’s recent work on concurrent programming with revisions and isolation types [5] provides a model in which programmers can declare what data they wish to share between tasks (threads), and execute tasks concurrently by forking and joining revisions. While a revision’s `rjoin` operation provides similar functionality to a world’s `commit` operation, there are some important differences. For example, unlike `rjoin`, the `commit` operation detects read-write conflicts (these result in a `CommitFailed` exception). Also, “revisions do not guarantee serializability ... but provide a different sort of isolation guarantee” and the authors “posit that programmers, if given the right abstraction, are capable of reasoning about concurrent executions directly.” [5]. While this may indeed be the case, we believe that serializability (which is supported by worlds) makes for a much more understandable programming model.

The idea of treating the program store as a first-class value and enabling programmers to take snapshots of the store which could be restored at a later time first appeared

in Johnson and Duggan’s GL programming language [18]. This model was later extended by Morrisett to allow the store to be partitioned into a number of disjoint “sub-stores” (each with its own set of variables) that could be saved and restored separately [22].

The main difference between previous formulations of first-class stores and worlds lies in the programming model: whereas first-class stores have until now been presented as a mechanism for manipulating *a single store* (or multiple partitions of the store, as is the case with Morrisett’s work) through a save-and-restore interface, worlds enable multiple versions of the store—several “parallel universes”—to co-exist in the same program. This makes worlds a better fit for “programming via experiments”, i.e., a programming style that involves experimenting with multiple alternatives, sometimes making mistakes that require retraction in order to make fresh starts. (This is difficult to do in mainstream imperative programming languages, due to the unconstrained nature of side effects.) It also makes it possible for multiple “experiments” to be carried out in parallel, which is something we intend to investigate in future work.

Tanter has shown that values that vary depending on the context in which they are accessed or modified—(implicitly) *contextual values*—can be used to implement a *scoped assignment* construct that enables programmers to control the scope of side effects [29]. Although Tanter’s construct does not support the equivalent of the commit operation on worlds, it is more general than worlds in the sense that it allows any value to be used as a context. However, it is not clear whether this additional generality justifies the complexity that it brings to the programming model. For example, while it is straightforward to modify a group of variables in the context of the current thread (e.g., thread id 382), or the current user (e.g., awarth), it is difficult to reason about the state of the program when both contexts are active, since they need not be mutually exclusive. (This is similar to the semantic ambiguities that are caused by multiple inheritance in object-oriented languages.)

Smith and Ungar’s *Us* [28], a predecessor of today’s Context-Oriented Programming (COP) languages [15], explored the idea that the state and behavior of an object should be a function of the perspective from which it is being accessed. These perspectives, known as *layers*, were very similar to worlds (they were first-class objects that provided a context in which to evaluate expressions) but did not support the equivalent of the *commit* operation.

Worlds enable programmers to enjoy the benefits of Functional Data Structures (FDSs) [24] without having to plan for them ahead of time. For instance, the example in Section 3.1 (Better Support for Exceptions) would require whatever data is modified in the `try` block to be represented as a FDS, which is inconvenient and potentially very time-consuming for the programmer. To make matters worse, the `try` block might call a function—in this case, the programmer would have to (non-modularly) inspect the code that implements that function in order to find out what data it may modify so that she can change it into a FDS. With worlds, none of this is necessary.

Lastly, a number of mechanisms for synchronizing distributed and decentralized systems (e.g., TeaTime [25/26] and Virtual Time / Time Warp [17]) and optimistic methods for concurrency control [21] rely on the availability of a rollback (or undo) operation. As shown in Section 3.2, a programming language that supports worlds greatly sim-

plifies the implementation of rollbacks, and therefore could be a suitable platform for building these mechanisms.

9 Conclusions and Future Work

We have introduced worlds, a new language construct that enables programmers to control the scope of side effects. We have instantiated our notion of worlds in `Worlds/JS` and `Worlds/Squeak` (extensions of JavaScript and Squeak Smalltalk, respectively), formalized the semantics of property/field lookup in these languages, and shown that this construct is useful in a wide range of applications. We have also described the efficient implementation strategy that was used in our Squeak-based prototype.

We believe that worlds have the potential to provide a tractable programming model for multi-core architectures. As part of the STEPS project [19,20], we intend to investigate the feasibility of an even more efficient (possibly hardware-based) implementation of worlds that will enable the kinds of experiments that might validate this claim. For example, there are many problems in computer science for which there are several known algorithms, each with its own set of performance tradeoffs. In general, it is difficult to tell when one algorithm or optimization should be used over another. Our hardware-based implementation should make it practical for a program to choose among optimizations simply by sprouting multiple “sibling worlds”—one for each algorithm—and running all of them in parallel. The first one to complete its task would be allowed to propagate its results, and the others would be discarded.

Also as part of the STEPS project, we intend to build a multimedia authoring system that supports “infinite” undo. Our bitmap editor and text editor (referenced in Section 3.2) are two distinct examples of undo, but we hope use worlds to implement a data model for all media types.

The top-level world’s commit operation, which is currently a no-op, might be an interesting place to explore the potential synergy between worlds and persistence. For example, a different implementation of `TopLevelWorld>>commit` that writes the current state of every object in the system to disk (to be retrieved at a later time) could be the basis of a useful checkpointing mechanism.

One current limitation of worlds is that they only capture the *in-memory* side effects that happen inside them. Programmers must therefore be careful when executing code that includes other kinds of side effects, e.g., sending packets on the network and obtaining input from the user. It would be interesting to investigate whether some of the techniques used in reversible debuggers such as EXDAMS [1] and IGOR [10] could be used to ensure that, for example, when two sibling worlds read a character from the console, they get the same result.

Acknowledgments. The authors would like to thank Gilad Bracha, Marcus Denker, Robert Hirschfeld, Mark Miller, Todd Millstein, Eliot Miranda, James Noble, Jens Palsberg, David Reed, Hesam Samimi, and the anonymous reviewers for their useful feedback, and Bert Freudenberg for writing the Slang code that enabled our Squeak primitives to query `IdentityDictionaries`.

References

1. Balzer, R.M.: EXDAMS—EXtendable debugging and monitoring system. AFIPS Spring Joint Computer Conference 34, 567–580 (1969)
2. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the scope of change in Java. In: OOPSLA 2005: Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 177–189. ACM Press, New York (2005)
3. Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A minimal module model supporting local rebinding. In: Böszörményi, L., Schojer, P. (eds.) JMLC 2003. LNCS, vol. 2789, pp. 122–131. Springer, Heidelberg (2003)
4. Bracha, G.: Monkey Patching (blog post) (2008), <http://gbracha.blogspot.com/2008/03/monkey-patching.html>
5. Burckhardt, S., Baldassion, A., Leijen, D.: Concurrent programming with revisions and isolation types. In: OOPSLA 2010 (October 2010)
6. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design rationale, compiler implementation, and applications. ACM Transactions on Programming Languages and Systems 28(3) (May 2006)
7. Denker, M., Gîrba, T., Lienhard, A., Nierstrasz, O., Renggli, L., Zumkehr, P.: Encapsulating and exploiting change with changeboxes. In: ICDL 2007: Proceedings of the 2007 International Conference on Dynamic Languages, pp. 25–49. ACM Press, New York (2007)
8. Ducasse, S., Lienhard, A., Renggli, L.: Seaside: A flexible environment for building dynamic web applications. IEEE Software 24(5), 56–63 (2007)
9. ECMA International. ECMA-262: ECMAScript Language Specification. European Association for Standardizing Information and Communication Systems, Geneva, Switzerland, third edition (December 1999)
10. Feldman, S.I., Brown, C.B.: IGOR: a system for program debugging via reversible execution. ACM SIGPLAN Notices 24(1), 112–123 (1989)
11. Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, functional pearl. In: ICFP 2002: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming, pp. 36–47. ACM Press, New York (2002)
12. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: POPL 2004: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 111–122. ACM Press, New York (2004)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional, Reading (1995)
14. Hårris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPOPP 2005: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 48–60. ACM Press, New York (2005)
15. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-Oriented Programming. Journal of Object Technology (JOT) 7(3), 125–151 (2008)
16. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: the story of Squeak, a practical Smalltalk written in itself. SIGPLAN Notices 32(10), 318–326 (1997)
17. Jefferson, D.R.: Virtual time. ACM Transactions on Programming Languages and Systems 7(3), 404–425 (1985)
18. Johnson, G.F., Duggan, D.: Stores and partial continuations as first-class objects in a language and its environment. In: POPL19'88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 158–168. ACM Press, New York (1988)

19. Kay, A., Ingalls, D., Ohshima, Y., Piumarta, I., Raab, A.: Proposal to NSF (2006), http://www.vpri.org/pdf/NSF_prop_RN-2006-002.pdf (granted on August 31, 2006)
20. Kay, A., Piumarta, I., Rose, K., Ingalls, D., Amelang, D., Kaehler, T., Ohshima, Y., Thacker, C., Wallace, S., Warth, A., Yamamiya, T.: Steps toward the reinvention of programming (first year progress report) (2007), http://www.vpri.org/pdf/steps_TR-2007-008.pdf
21. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. *ACM Transactions on Database Systems* 6(2), 213–226 (1981)
22. Morrisett, J.G.: Generalizing first-class stores. In: *SIPL 1993: Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, pp. 73–87 (1993)
23. Nandivada, V.K., Jagannathan, S.: Dynamic state restoration using versioning exceptions. *Higher-Order and Symbolic Computation* 19(1), 101–124 (2006)
24. Okasaki, C.: *Purely functional data structures*. Cambridge University Press, Cambridge (1999)
25. Reed, D.P.: *Naming and synchronization in a decentralized computer system* (PhD dissertation). Technical Report TR-205, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)
26. Reed, D.P.: Designing Croquet’s TeaTime: a real-time, temporal environment for active object cooperation. In: *OOPSLA 2005: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 7–7. ACM Press, New York (2005)
27. Shavit, N., Touitou, D.: Software transactional memory. In: *PODC 1995: Proceedings of the 12th Annual ACM Symposium on Principles of Distributed Computing*, pp. 204–213 (1995)
28. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. *Theor. Pract. Object Syst.* 2(3), 161–178 (1996)
29. Tanter, E.: Contextual values. In: *DLS 2008: Proceedings of the 2008 Dynamic Languages Symposium*, pp. 1–10. ACM Press, New York (2008)
30. Warth, A.: OMeta/JS website, <http://www.tinlizzie.org/ometa-js/>
31. Warth, A.: *Experimenting with Programming Languages*. PhD dissertation, University of California, Los Angeles (2009)
32. Warth, A., Piumarta, I.: OMeta: an Object-Oriented Language for Pattern-Matching. In: *OOPSLA 2007: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, ACM Press, New York (2007)
33. Warth, A., Stanojević, M., Millstein, T.: Statically scoped object adaptation with Expanders. In: *OOPSLA 2006: Proceedings of the 21st ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 37–56. ACM Press, New York (2006)

Can We Avoid High Coupling?

Craig Taube-Schock¹, Robert J. Walker², and Ian H. Witten¹

¹ University of Waikato, Hamilton, New Zealand

² University of Calgary, Calgary, Canada

Abstract. It is considered good software design practice to organize source code into modules and to favour within-module connections (*cohesion*) over between-module connections (*coupling*), leading to the oft-repeated maxim “low coupling/high cohesion”. Prior research into network theory and its application to software systems has found evidence that many important properties in real software systems exhibit approximately *scale-free structure*, including coupling; researchers have claimed that such scale-free structures are ubiquitous. This implies that *high coupling must be unavoidable*, statistically speaking, apparently contradicting standard ideas about software structure. We present a model that leads to the simple predictions that approximately scale-free structures ought to arise both for between-module connectivity and overall connectivity, and not as the result of poor design or optimization shortcuts. These predictions are borne out by our large-scale empirical study. Hence we conclude that high coupling is *not* avoidable—and that this is in fact quite reasonable.

1 Introduction

We have long heard the maxim of “high cohesion/low coupling” in software design. It is generally believed that high coupling—that is, high levels of between-module connectivity—particularly signals poor design, as it leads to greater difficulties in modification, comprehension, and parallel development [31,2,28,32,38,17]. Unfortunately, while software can be poorly created with definitely excessive coupling, it is not immediately clear whether high coupling *can* be definitively eliminated in all circumstances.

Analysis of complex networks has revealed the presence of (approximately) *scale-free structure* in networks from a wide variety of fields [27]. A scale-free network is one that has a power-law degree distribution [1,3], characterized by having a majority of nodes involved in few connections and a few nodes involved in many connections. Roughly speaking, an approximately scale-free distribution would manifest itself as a straight line on a log–log plot of the connection degree histogram (i.e., number of connections vs. frequency of nodes). Evidence suggests that other distributions produce similar network characteristics [19,8] and that these distributions exist for various relations in procedural and object-oriented software systems [37,26,24,30,35,4,18,15,17,9,23,6,16,14,36,12]. This set of distributions are known as *heavy-tailed* to differentiate the decay characteristic of their probability mass function from that of typical exponential decay;

a significant probability of occurrence exists even at several standard deviations above the mean [8].

While the application of complex networks to the analysis of software has seen much work, it overlooks a surprising consequence of the observed phenomena:

The presence of any heavy-tailed distribution describing the degree counts in the connectivity network means that there must be nodes that are highly coupled, and even nodes that are very highly coupled, relative to the mean level for each system.

Given the specific distributions observed empirically, highly coupled nodes are commonplace even in an absolute sense.

This is a paradox. We are all well aware that high coupling is something to be avoided. Yet, the mechanics of heavy-tailed distributions are such that—should they be found to be as universal as claimed—high coupling apparently *must* be present. Possible immediate explanations present themselves: perhaps the systems from which the empirical data were collected were poorly designed and thus the conclusions not indicative of good practice; perhaps the researchers made some serious mistake in the data collection or analysis, such as not actually looking at “real” coupling; perhaps we are witnessing an effect from well-known, practical optimizations; perhaps we have to accept that high coupling is actually *necessary* after all.

To resolve this paradox, we begin (in Section 2) by examining the previous work in network theory, especially as it has been applied to software systems; we find some weaknesses in the generality of the empirical results and a few questionable premises and conclusions, but nothing so serious as to resolve the paradox. Next, we present (in Section 3) a model, based largely on existing ideas, for why (approximate) scale-free structure *should* arise in overall connectivity and thus more specifically in between-module connectivity. These simple predictions are then tested in an empirical study, run against 97 open source software systems (the Qualitas Corpus [33]) written in the Java programming language, across granularities ranging from the statement level to the package level. The design of the study and the experimental apparatus are described in Section 4, while the results are analyzed in Section 5. Remaining issues and observations are discussed in Section 6.

This paper makes three contributions: (1) a demonstration that overall connectivity follows a heavy-tailed distribution across the spectrum of granularity for a large number of open-source systems—regardless of maturity, degree of active support, and level of use; (2) a demonstration that between-module connectivity ubiquitously follows a heavy-tailed distribution—and thus highly-coupled nodes are ubiquitous; and (3) an explanatory model as to why having some areas of high coupling is consistent with good software design practices.

2 Scale-Free Structure and Its Application to Software

There has been considerable interest in the study of the structure of networks [27], due especially to the observation that networks derived from “real”

phenomena (as opposed to phenomena derived from the simulation of mathematical models) have a degree distribution that follows a *power law* [1,3,13]. This is in contrast to networks generated using the algorithmic techniques defined by Erdős and Rényi, which possess a Poisson degree distribution [11].

2.1 Power-Lawdistributions

In networks that possess a power-law degree distribution, the probability that a node x has the degree $\text{deg}(x)$ is proportional to $\text{deg}(x)^{-\alpha}$ where $\alpha > 1$: i.e.,

$$p(\text{deg}(x)) = C \text{deg}(x)^{-\alpha}, \quad (1)$$

for some normalization constant C chosen to satisfy $\sum_{y=1}^{\infty} C y^{-\alpha} = 1$ (because of the definition of probability mass function). In most power-law distributions encountered in practice, $2 \leq \alpha \leq 3$, but this is not always the case [8]. From such distributions, two key connectivity characteristics emerge:

1. The mean connectivity is low relative to the range because the distribution is left-skewed. This indicates that most of the nodes in the system have low connectivity.
2. The range of connectivity has the potential to be several orders of magnitude greater than the mean, depending on the size of the network. Thus, nodes will be present that exhibit high degrees of connectivity with respect to the mean; these nodes will reside in the *heavy tail* of the distribution.

Networks with a power-law degree distribution are called *scale-free* [1,3], due to the fact that they are self-similar at “all” scales.

Transforming Equation 1 to take the logarithms of each side, we arrive at:

$$\log(p(\text{deg}(x))) = -\alpha \log(\text{deg}(x)) + \log(C), \quad (2)$$

which presents itself as a straight line on a log–log plot of $\text{deg}(x)$ versus $p(\text{deg}(x))$ (practically, the frequency observed in empirical data). One is thus tempted to perform a linear regression to the log–log plot to determine the parameters of the model. Strictly speaking, this is not a statistically valid procedure for a variety of reasons [8], not least of which is the fact that data drawn from many different distributions can lead to a roughly straight line on a log–log plot. For our purposes, it is enough to note that any of these *heavy-tailed distributions* lead to an inevitable consequence: the probability is surprisingly large that there exist data points in the heavy tail that are multiple standard deviations away from the mean. The lack of such points would actually invalidate the claim that the data follows a heavy-tailed distribution, as an approximately straight line would not be observed on the log–log plot.

It is well-observed [8] that, for values below some threshold $\text{deg}(x) = d_{\min}$, the power law breaks down because there is some minimum natural scale preventing the behaviour from continuing all the way to 0.

2.2 Empirical Findings

There have been several investigations into the structure of software systems that have revealed the presence of power-laws and other heavy-tailed distributions. Wheeldon and Counsell [37] examined power-laws in the class coupling relationships within 3 industrial systems for the purpose of using power-law distributions to predict coupling patterns. They examined 5 different class-coupling relationships (inheritance, interface, aggregation, parameter type, and return type) and concluded that not only does each have a power-law distribution but the relationships are independent of each other. Wheeldon and Counsell do not include coupling as a result of method invocation, and no analysis occurs below the class level.

Myers [26], Marchesi et al. [24], Potanin et al. [30], and Gao et al. [12] observed power-laws in both the in-degree and out-degree distributions of modules in a total of 26 different software systems. Baxter et al. [4] examined 56 systems—many of which are also contained in the Qualitas Corpus [33]—for a large set of measures including some coupling measures, but considered them independently from one another. They observed log-normal out-degree distributions, and some specific coupling measures did not match a heavy-tailed distribution in some instances, perhaps hinting at a lack of universality. Jing et al. [18] found power-laws in the measures weighted methods per class (WMC) and coupling between objects (CBO) for 4 open-source software systems. Concas et al. [9] examined 10 properties of 3 software systems and found those properties to have both Pareto and log-normal distributions. Ichii et al. [16] examined 4 measures (including two variants of WMC) on 6 systems, finding that in-degree follows a power law while out-degree follows some other heavy-tailed distribution. Louridas et al. [23] found power-laws present in the dependencies of software libraries, applications, and system calls in the Linux and FreeBSD operating systems and concluded that power-laws are ubiquitous in software systems.

None of the aforementioned investigations considered software systems at the level of statements and variables, limiting the generality of the findings. Some of the investigations did not explicitly plan to investigate coupling. Myers [26] considered only inheritance and aggregation relationships. Concas et al. [9] focused mostly on size measures, but did include a count of method invocations between classes, which they found to conform to a power-law; however, they did not examine other forms of coupling. Gao et al. [12] considered method–method interaction, thereby excluding other class-level coupling measures.

Hyland-Wood et al. [15] examined coupling relationships at differing levels of granularity (package, class, and method level, but not statement level) for 2 separate open source projects over a 15 month period and concluded that scale-free properties were present at all levels of analysis for each snapshot although they note that these properties were approximate in most cases. While demonstrating the relationship of scale-free structure between differing levels of granularity, this study's lowest level of analysis was that of methods.

Vasa et al. [36] noted that many software metrics have a skewed distribution, which makes the reporting of data using central tendency statistics unreliable.

To address this, they recommend adopting the use of the *Gini coefficient*, which has been used in the field of economics to characterize the relative equality of distributions. They examined 46 systems on a variety of measures, where two of the measures are related to coupling (in-degree count and out-degree count). Their findings appear to mimic the findings of Myers [26] and Gao et al. [12] that in-degrees and out-degrees have differing distributions. However, their findings do not address the structure of software at the source code level.

Some of the investigations had confounding factors, which makes them difficult to directly compare with our investigation. Marchesi et al. [24] examined classes in Smalltalk systems, but issues of dynamic binding prevented precise resolution of between-module interactions. To circumvent these issues, dependency relationships that could only be resolved at runtime were approximated using a weighting function, but it is not clear what effect this transformation may have had. Potanin et al. [30] investigated object graphs, which are not directly comparable to class graphs. For example, collection objects may have large numbers of runtime associations that would not be detectable through static analysis. Similarly, the number of instances of each class could skew the total degree distribution as classes with higher numbers of instances would have greater weight in the analysis. It is not clear that scale-free structure in an object graph translates to scale-free structure in its corresponding class graph.

Valverde et al. [34] and Jenkins and Kirk [17] note that nodes with large numbers of dependencies (termed *hubs*) fall in “the set of bad design practices known as antipatterns” [20]; they fail to identify that the ubiquitous presence of heavy-tailed distributions implies the presence of hubs.

2.3 Process Models Leading to Scale-Free Structure

To offer an explanation as to how a power-law could develop, Barabási and Albert [3] considered the evolution of complex networks as they increased in size and noted that the *preferential attachment* model caused scale-free structure to emerge. In this model, newly added nodes preferentially attach to nodes that have been in the network the longest time, resulting in a structure where most nodes have limited connectivity and only the oldest have high connectivity.

Several criticisms of the preferential attachment model have been put forth, especially as it applies to software systems. Valverde et al. [34] complain that “no design principle explicitly introduces preferential attachment, nor scaling”, offering an alternative model based on optimizing designs to minimize the path length between nodes. Unfortunately, their complaint about design principles is largely irrelevant since known design principles are rules of thumb and incomplete. Furthermore, their evaluation is based on the assumption that the systems they look at possess optimal designs—because they have been under development for a long time. This contradicts Lehman’s Law of Declining Quality [21] and the community’s general experience.

Myers [26] dismisses preferential attachment because it cannot generate the hierarchical structures present in software; he suggests that scale-free structure arises instead from continuous refactoring. But not all software undergoes

non-trivial refactoring, so either his model is false or we would expect there to exist software systems that do not exhibit scale-free structure—he analyzed only 6 industrial systems that had been under development for prolonged periods and hence could be assumed to have undergone at least some refactoring. Keller [19] points out that many different processes can lead to scale-free structure, and that in fact, the necessary constraints are quite meagre.

Jenkins and Kirk [17] state “preferential attachment relies on newly added nodes having prior knowledge of the rest of the network, which seems implausible, since software is built in pieces from a series of sources using various rules for design patterns which do not apply to the finished software graph”—a clearly untenable assertion, since the developer must have prior knowledge of the network in order to select to which parts of it a newly added node should connect.

Chen et al. [6] added a factor to the preferential attachment model that made it less likely that attachment would happen to a node in another module; they fail to consider how modules themselves are added, deleted, or refactored within a system, and they only validate their conclusions against a single (albeit large and important) system. Li et al. [22] accept the preferential attachment model wholeheartedly without addressing Myers’s concern that it fails to explain hierarchical structures; they evaluate their conclusions on two systems.

3 Model

It is generally accepted that dependency between programmatic entities within a software system has a direct impact on that system’s ease to be changed, understood, and developed in parallel, and that a key indicator of dependency is connectivity between entities [31,2,10,28,32,7,5]. In Section 3.1, we examine background on the interplay between connectivity and evolvability. We use this background to develop a model, in Section 3.2, for why overall connectivity should be expected to possess an approximately scale-free structure. Adding considerations of practical limitations on module sizes leads us to the conclusion that between-module connectivity should also possess an approximately scale-free structure—and thus, that highly-coupled entities must exist in any sizeable system.

3.1 Connectivity and Evolvability

Different theoretical models of the relationship between dependency and evolvability were developed by Simon [31] and Alexander [2] (Alexander used the term *adaptation*). Simon focused on the structures common to all complex systems while Alexander focused on the design of systems intended to fit a particular problem. Both researchers viewed complex systems as sets of “components” (we will use the term *entities* or *nodes* to avoid further overloading the term “component”) that are organized in a hierarchical structure, which interact in a *non-simple* way. Both viewed the evolvability of complex systems as a probabilistic function based on the interdependency of entities.

In these models, the evolvability of a system is a function of its stability with respect to change propagation. To illustrate this point, consider Figure 1(a). Nodes labelled 1, 2, and 3 share mutual dependency due to their structure of connectivity. Should one of the nodes change, the probability of that change propagating through a connection with dependent nodes is determined by a probability distribution function that could, in principle, be determined empirically. Change propagation may necessitate further change, and so on, thereby increasing the number of structural modifications, which increases the time necessary for the system to stabilize. Such change propagation is often called a *ripple effect* [32,38]. Overall system stability is a function of the probability of propagation p and the number of pathways through which propagation can occur.

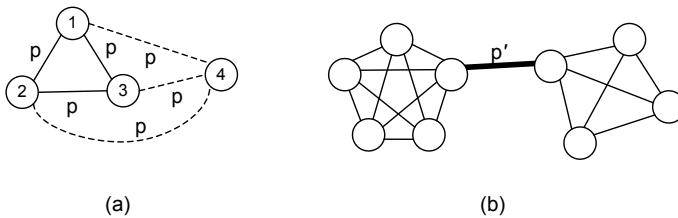


Fig. 1. Evolvability: (a) change propagation; (b) clustering to minimize propagation

As a system increases in size, the potential for instability caused by ripple effects also increases. In Figure 1(a), the introduction of node 4 introduces three potential new pathways through which change propagation may occur. Since systems require interaction between entities to function, limiting the number of entities, or limiting their ability to interact, may increase a system's stability at the cost of limiting its capabilities. To mitigate the effects of dependency while allowing a system to increase in size requires the use of modularization. Figure 1(b) shows entities grouped into two modules. Entities within each one are free to interact, but interaction between modules is strictly controlled. This structure increases the overall stability of the system by attempting to contain ripple effects within module boundaries.

Whether or not modularization mitigates overall change propagation depends on the probability of propagation between modules (p' in Figure 1(b)). Alexander noted that, in a complex system, the strength of connectivity between entities is not homogeneous [2]. Because of this, he specified that module boundaries be chosen in a way that places entities for which change propagation is high in the same module. This structure minimizes the probability of change propagating between modules.

The principles outlined by Simon and Alexander are echoed in later work by Dijkstra [10] and by Parnas [28,29]. Parnas compared two software systems that were written to address the same problem. The key difference between the systems was not the selection of individual entities, but rather the criteria used

to construct module boundaries. In the first system, modules were constructed by identifying key steps in the overall processing, while for the second they were constructed by using *information hiding* as the primary criterion. The modularizations created in the first system resulted in the sharing of variables between modules. Since the probability of propagation is high between processing statements and variables, shared variables act as a bridge through which change propagation flows between modules. Building module boundaries based upon information hiding, however, encapsulated the interaction between processing statements and variables within modules, thereby constraining the flow of change propagation between modules. Parnas argued that the evolvability (along with the understandability and the ability to construct the modules independently) was higher in the second system due to the structure of its modules.

Recognizing the effects of between-module interaction on evolvability, Stevens et al. [32] coined the term *coupling*. In a modularized system, coupling is defined as “the measure of strength of association established by a connection from one module to another” [32, pp. 233]. In Parnas’s example, modularization of the second system exhibited lower coupling than the first and it was therefore deemed to be more evolvable.

3.2 Scale-Free Structure in Overall Connectivity

We adopt the preferential attachment model as the starting point for our model, and initially ignore considerations of constraining the maximum allowable connectivity. Barabási and Albert [3] base the notion of preferential attachment on an evolutionary process, in which the probability of attachment increases simply because a node has been in the network longer. We can translate this into more appropriate selection criteria at play in software development that should result in the same overall effect. (1) The probability of attachment will be directly proportional to the usefulness of the functionality provided. The general usefulness of nodes can be expected to vary quite widely; in fact, a node that has proven generally useful in the past is more likely to be generally useful in the future. (2) A developer has to be aware of existing functionality to make use of it. The most commonly used functionality in a system is most likely to be familiar to that developer, his co-workers from whom he is likely to seek help, or any on-line documentation or examples he is likely to encounter. (3) A developer is more likely to use functionality in which he has greater trust, because he or others have used it a lot in the past, or because it has been actively supported for a prolonged time and has acquired a reputation for quality.

Two issues might skew empirically-derived distributions. First, below some minimal scale, insufficient nodes exist for the Law of Large Numbers to hold. Second, developers make errors (contrary to the assumptions of Valverde et al. [34]); a developer might add a spurious connection or fail to add a necessary connection. Whatever the distribution from which these errors would be drawn, their probability is necessarily much lower than that of the correct nodes, so the result would be a noisy scale-free structure. As a result, we arrive at the following hypothesis:

Hypothesis 1*: The overall connectivity network of source code entities for any software system above some minimum size follows an (approximately) scale-free distribution, when no constraints are externally applied to the maximum level of overall connectivity.

Now, we must consider the effect of disallowing any entities to be added to the system with connectivity greater than some value d_{\max} . Imagine that, through the standard preferential attachment model, we obtain the first entity e that would normally (in the absence of the constraint) have connectivity of $\deg(e) = d > d_{\max}$. Simply discarding this e is not an option—it was presumably to be added to serve a new purpose within the system. Therefore, we must replace e with some alternative that satisfies the constraint. We can begin by ignoring the constraint and nevertheless insert e into the network, then transform (refactor) the network to again support the constraint.

To replace e , two or more other entities e_i could take its place, each of which (at best) would inherit an independent portion of the connections of e ; each of these replacement entities would need at least one connection with another of the replacement entities but as many as one connection with every of the replacement entities. Thus, we have $\deg(e_1) = p_1d + \hat{p}_1n$, $\deg(e_2) = p_2d + \hat{p}_2n$, \dots , $\deg(e_n) = p_nd + \hat{p}_nn$ where p_i is the fraction of the connections of e inherited by e_i and \hat{p}_i is the fraction of the replacement nodes to be connected to e_i . If any of these replacement entities themselves fail to respect the constraint, the process can recurse. To ensure that this replacement process halts, we can add an additional, simple constraint: that $\deg(e_i) < \deg(e)$ in all cases; thus, progress is made at each iteration and eventually the constraint is satisfied.

To determine specifically to which other entities e_i will be connected, we can return to the original principle of preferential attachment. But preferential attachment is known to result in a scale-free structure in the limit of long time (equivalently, large number of network evolution steps). Thus for any arbitrary d_{\max} and the simple requirement for progress at each replacement step, *a connectivity network with a constraint on its maximum degree will also result in a (different) scale-free structure*. Thus we can revise Hypothesis 1* to eliminate the clause regarding maximum connectivity not being constrained:

Hypothesis 1: The overall connectivity network of source code entities for any software system above some minimum size follows an (approximately) scale-free distribution.

If “high coupling” is defined in terms of number of standard deviations away from the mean, there will thus remain highly coupled entities (ignoring the question of between-module versus within-module connectivity for the moment) after the replacement process—even though the maximum absolute coupling level will have been reduced. However, presumably any arbitrary $d_{\max} \geq 1$ is achievable, and at some point, d_{\max} would be considered “low enough” for practical purposes; thus, “high coupling” would not be universal according to a more absolute definition. The question then becomes: are there other negative consequences of

the replacement process that would tend to prevent an arbitrarily low d_{\max} from being achieved? If so, high coupling would remain, even when defined in absolute terms.

To address this question, consider *inlinks* (inbound connections) and *outlinks* (outbound connections) for all source code entities. In general, outlinking is constrained to be reasonably small. For example, class declarations have few direct superclasses and directly implement few interfaces. Variables are of a single type (or few types in the case of generics) and individual statements tend to be limited to the number of variable access or method invocations due to practical issues, such as style guidelines and the difficulty of reading statements that extend beyond a programmer's screen width. Method declarations have practical limits on the number of return types, the number of parameters, and the number of exceptions that can be thrown by the method. There is, however, no constraint on the number of inlinks that can be made to an entity that has a name within a defined scope. Classes can be used in any number of variable declarations and methods can be invoked from any number of statements. Variables, too, can be used in a variety of different contexts although they will tend to be limited to a stricter scope than that of classes and methods. For these reasons, high connectivity is largely due to inlinks.

A source code entity that exhibits high connectivity is thus likely to do so because of its utilization in multiple contexts. Indeed, use in multiple contexts is a direct side effect of hierarchical structure. Consider a source code entity e such that $\text{deg}(e) > d_{\max}$ and for which the number of connections is largely due to inlinking. To replace e by two or more entities (e_i) in an attempt to satisfy $\text{deg}(e_i) \leq d_{\max}$ would require that all the replacement nodes e_i provide the same utility as e . This suggests the introduction of code clones, which is considered to be poor design. The ability to reuse source code entities suggests that an arbitrarily low d_{\max} cannot be practically achieved.

3.3 Scale-Free Structure in Between-Module Connectivity

We still have to deal with Myers's concern about preferential attachment being an unsuitable model for software evolution because it does not generate hierarchical structures [26]. While hierarchical structure does not emerge from the preferential attachment model, Myers's analysis does not consider a variation on preferential attachment for which hierarchy is imposed by some external means (such as programming language grammar). As new nodes are added to the network, they will minimally link to a parent node. The programming language syntax and semantics impose constraints on which nodes may act as an acceptable parent based on the type of the node being added. For example, the Java programming language only allows method declarations to be placed within the source code graph as children of a class declaration. This constrains node linkages in a preferential manner, although the preferential probability function differs from that defined in the preferential attachment model [6].

A final question remains: is it reasonable to consider that scale-free structure for overall connectivity necessarily leads to high coupling? Consider the models of Alexander [2] and Simon [31], discussed in Section 3.1. A primary function of modularization is to minimize propagation of change between modules, and to this end, propagation of change within a module can be ignored if the module stabilizes quickly. Module stabilization time is largely affected by limiting module size. Having fewer entities within a module reduces the number of pathways through which change propagation can occur, thereby resulting in pressure to limit the size of modules.

For any source code entity that exhibits high connectivity its links will resolve to other entities contained either in the same module or a different module. If they are resolved within a module, this implies that there are enough entities within the module with which resolution can occur, and that suggests a large module if connectivity is high. Since there is pressure to limit the size of modules, this suggests that high connectivity of source code components is resolved between modules, which represents a form of high coupling. Thus we arrive at:

Hypothesis 2*: The between-module connectivity network of source code entities follows a heavy-tailed distribution.

But the same replacement process can be applied to between-module connectivity as was for overall connectivity, with the same constraints. Thus, while between-module connectivity can be reduced, it cannot be practically reduced beyond some minimum level. We can therefore adjust Hypothesis 2*:

Hypothesis 2: The between-module connectivity network of source code entities follows a heavy-tailed distribution, and the degree of left skewness has some maximum level.

Hypothesis 2 (if supported) implies that highly-coupled entities *must* exist for a sizeable system, even when considered in absolute rather than relative terms.

4 Empirical Study

This study comprises an empirical investigation of source code connectivity as observed in practice. The empirical data comes from the Qualitas Corpus [33], a collection of 100 independent open-source software systems written in the Java programming language. The corpus contains at least one version of each independent system, and for some systems multiple versions are present. Since different versions of the same software are not independent, our study only includes one version of each system, specifically the latest one within the corpus, resulting in 100 systems available for study. For three of the systems (eclipse_SDK-3.3.2-win32, myfaces_core-1.2.0, and jre-1.5.0_14-linux-i586), source code was absent from the corpus and thus discarded from the examination set, leaving 97 systems for investigation.

Table 1 provides a truncated view of the systems examined. For each one, counts are reported for: source code entities, connections between entities, modules, classes, methods, statements, and variables. Source code entities include

Table 1. Structural measures of the systems that were examined. “Nod” = nodes; “Cnx” = connections; “Mod” = modules; “Cls” = classes; “Mth” = methods; “Blk” = blocks; “Sta” = statements; “Var” = variables

#	Name/Version	Nod	Cnx	Mod	Cls	Mth	Blk	Sta	Var
1	derby-10.1.1.0	318831	809952	135	1805	25067	56357	160555	74910
2	gt2-2.2-rc3	256838	651522	219	3453	26347	52556	106738	67523
3	weka-3.5.8	248704	682151	91	2019	19169	47561	124152	55710
4	jtopen-4.9	230394	593240	18	1940	20559	42206	112259	53410
5	tomcat-5.5.17	177249	433523	149	1777	17152	36247	80214	41708
6	compiere-250d	155379	388859	43	1260	18128	25458	73472	37016
92	jmoney-0.4.4	6310	17618	6	193	713	996	2989	1411
93	nekohtml-0.9.5	6606	17153	7	54	422	1453	2887	1781
94	jchempaint-2.0.12	5757	15844	8	125	419	1146	2696	1361
95	jasml-0.10	5482	15419	8	53	256	895	3011	1257
96	fitjava-1.1	3862	10296	5	96	462	786	1564	947
97	picocontainer-1.3	3771	9117	5	99	540	842	1155	1128

modules, classes, method declarations, blocks, statements, and variables; each is modelled as a node within a directed graph. Connections between source code entities are modelled as links between nodes; these include parent/child relationships, method invocations, superclass/subclass relationships, superinterfaces, type usage, variable usage, and polymorphic relationships. The systems shown in Table 1 are sorted in descending order by node count; only the top and bottom six systems are presented.

4.1 Graph-Based Source Code Representation

The basis of our analysis is a directed graph representation of source code, where nodes represent source code entities (packages, classes, methods, blocks, statements, and variables), and links (directed arcs) represent connections between entities (hierarchical containment, method invocation, superclass, implementation, type, variable usage, and method overriding). Figure 2 shows the meta-model used in this investigation, similar to that of Mens and Lanza [25].

Our model differs mostly in terms of the level of details provided (Mens and Lanza’s metamodel is language-independent and was simplified for readability); however, there are three key structural differences. (1) Our model explicitly defines package and block entities, which are implicit in Mens and Lanza’s model. Our reasoning for inclusion of these structural features is that they are important means of structuring in practice, and they could have a significant effect on the connectivity network. (2) Our model is more explicit about containment and hierarchical structure. For example, classes can contain other classes and statements can contain other statements or blocks (such as the code to be executed as part of a loop). This kind of containment definition is particularly relevant in terms of variable declarations. Our model allows for explicit containment of variables within classes/interfaces (as class and instance variables), method

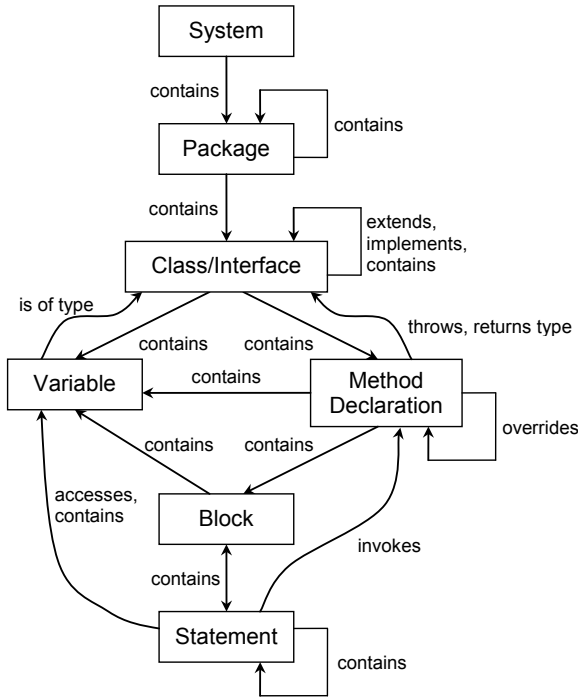


Fig. 2. Metamodel applied in our analysis

declarations (as parameters), blocks (as scope-limited variables), and statements (e.g., a for-loop counter), where the Mens and Lanza model appears to be focused exclusively on instance variables. (3) Our model supports relationships that do not exist in Mens and Lanza’s model. For example, variables have a type (which in our model is represented as a relationship between the variable declaration entity and the associated type entity) and methods have a return type and can specify exceptions that can be thrown from within the method’s body. Method invocation is represented as a relationship between a statement and the target method declaration, whereas the Mens and Lanza model represents invocation as an entity contained within a method.

To illustrate the use of this metamodel, we provide a Java source example (Figure 3) and the resulting directed graph (Figure 4). In Figure 4, different node categories are illustrated as different shapes and link categories are shown using different colour and line styles. To improve readability, some relationships shown in the metamodel are excluded from the example (specifically “extends”, “implements”, “throws”, and “overrides”). The example source code contains two class definitions (X and Y), which are represented as Type nodes in the graph, and each class is located within separate packages, p1, and p2. The code contains several references to int, a primitive data type in Java. The code contains three method declarations—m1(), getVar2(), and getValue()—the first two being

```

package p1;
public class X {
    private Y var1;
    private int var2;
    public int m1() {
        int temp;
        temp = var1.getValue() + getVar2();
        return temp;
    }
    private int getVar2() {
        return var2;
    }
}

package p2;
public class Y {
    public int getValue() { ... }
}

```

Fig. 3. Sample Java source code listing

defined within class *X* and the last within class *Y*. To simplify this example, the implementation is provided for only two of the methods.

The hierarchical structure¹ of the source code is maintained through *Parent* links. Child nodes connect to their parent source code entities, and each node can only have one hierarchical parent. For example, class *X* is in package *p1*, and method *m1()* is defined within class *X*. Specification of type (as is seen in variable declaration and method return type specifications) is represented as a link from the specifying node to the type declaration node. In the example, instance variable *var1* (contained in class *X*) has a reference to class *Y*, creating a link between node *var1* and node *Y*. Similarly, each of the methods are declared to have an *int* return type, so there is a link from each method to the node representing the *int* type. There are two method invocations, which are represented as links between the calling statement node and the called method declaration node. Finally, uses of variables (*var1*, *var2* and *temp*) are represented as links between the using statement and the used variable declaration node.

This structure is constructed through the standard parser of the Eclipse integrated development environment.² The Eclipse parser is robust to all versions of the Java programming language and because it is used in both industrial-strength toolsets and research environments, it provides a reasonable basis upon which we conduct this investigation. The abstract syntax tree (AST) for each compilation unit in a given system is constructed using the parser. The hierarchical relationships in the resulting directed graph are derived from the

¹ Note that, in this context, “hierarchical” refers to the syntactic hierarchy, and should not be confused with the type hierarchy.

² <http://www.eclipse.org>

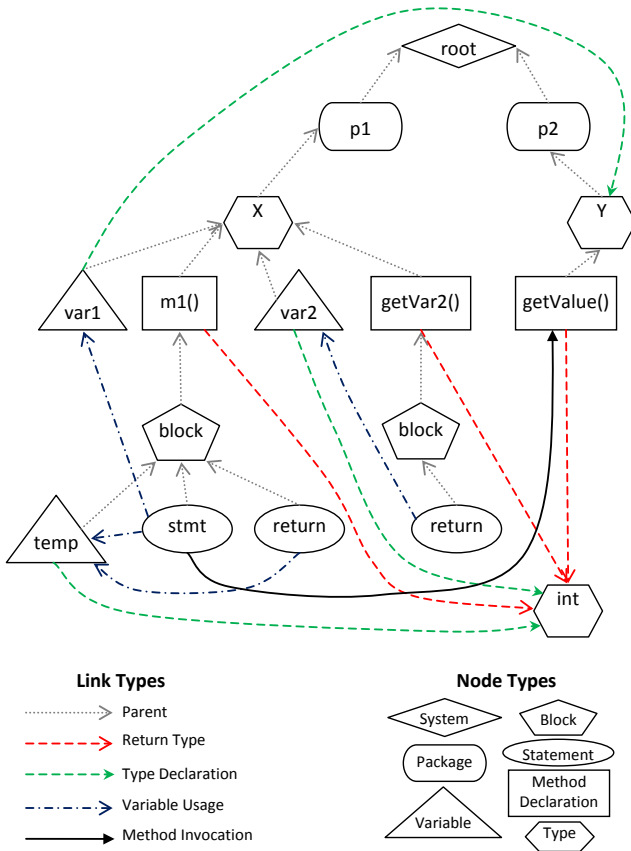


Fig. 4. Directed graph of the sample source code

hierarchical structure of the AST. Each compilation unit is inserted into the hierarchy, according to package definition, under a root node that represents the whole system. The remaining links are derived from the semantic information contained in the AST. In the case of type relationships, the Eclipse parser provides the fully qualified name of all resolved types, which is used to resolve the target node within the directed graph (the source node is implicit to the context of the type relationship). In the case of method invocations, special care is needed to correctly resolve overloaded methods as the fully-qualified names for overloaded methods are the same. To resolve this ambiguity, we extract the fully qualified name as well as the declared method’s signature for each method declaration (which are guaranteed to be unique within the containing class). This signature is used in conjunction with the method’s fully qualified name to resolve to the correct method declaration node. Finally, the Eclipse parser cannot provide a fully qualified name for variables that are embedded within blocks because blocks do not provide a namespace for contained variables. However, because the hierarchical structure of the source code is preserved in our

directed-graph representation, the scope of any node can be computed. In the case of variable accesses for which the parser has not provided a fully qualified name, the scope of the accessing node is computed and the variable in question is resolved within that scope by unqualified name.

We consider package declarations, class and interface declarations, method declarations, variable declarations, blocks, and statements as the base units of analysis. The form of other substructures, such as expressions and subexpressions, are highly constrained by the syntax of the language, rather than necessarily taking on a form that arises more naturally; thus, we ignore them for the sake of this analysis. To eliminate subexpressions from the data structure, all links for each subexpression are collapsed into the nearest non-expression ancestor node within the structure. For example, the statement `return x * y;` is represented by four nodes in the AST: the return statement, the multiplication expression, and the two variable references. Nodes `x` and `y` have links to the associated variable declaration nodes and the node representing the multiplication has no links (other than hierarchical containment ones). The relationships between `x` and `y` and their respective variable declarations are collapsed into the return statement. Once all subexpression relationships are resolved and collapsed, subexpression nodes are removed from the data structure.

Embedded within the source code structure are polymorphic relationships that are not explicitly identified by the compiler. Specifically, polymorphic method invocation is resolved at runtime: it is a dependency relationship that is implicit within the inheritance structure defined by superclass and subclass relationships. To make this relationship explicit, all overriding method declarations are identified and a link is added between each declaration and all ancestor method declarations in the class and interface hierarchy that have the identical signature. The Java programming language allows for single inheritance of classes but implementation of interfaces, which is supported by our toolset.

Virtually all software systems contain references to externally defined entities (e.g., libraries and programming language types). Since external entities are not part of the system under investigation, they are not considered in the analysis. However, proxy nodes that represent external entities are included in the analytic structure to act as placeholders, thereby allowing consideration of the connections between internal and external entities. For example, variables of type `int` possess a link to an `int` type declaration node even though the `int` type is external to the software system.

4.2 Identification of Within-Module and Between-Module Links

In object-oriented programming, the key module in a software design is the object, and objects are represented in source code by their classification (`class`). For the purposes of this analysis, we consider `class` to be the defining aspect of modular boundaries; alternatives are both possible and desirable targets for analysis, which we discuss further in Section 6. Identification of within-module and between module links is a matter of identifying links that cross class boundaries.

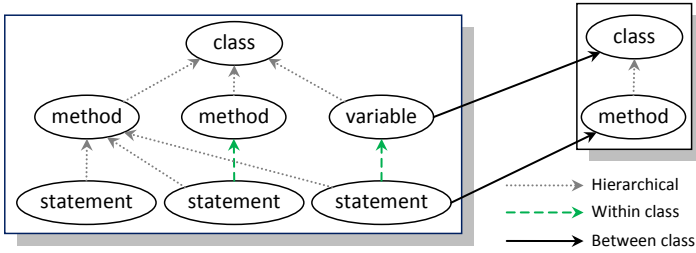


Fig. 5. Identifying within-module and between-module dependency

Encapsulated within the analytic structure are the hierarchical relationships for each entity in a software system. Within-module links are those for which both its associated source and target nodes share a common class in their hierarchical ancestry. Figure 5 illustrates examples of within-module and between-module links.

Hierarchical relationships introduce a confounding factor into our analysis. These links do not define relationships of direct interaction and because they are overwhelmingly within-module, their inclusion as part of the analysis will skew any comparison in that direction. For this reason, hierarchical links are used to identify structural boundaries that are relevant to the analysis but are not included as part of the computation of degree distributions.

Heavy-tailed distributions are then identified and fit to a power-law model via the informal procedure described in Section 2.1.

5 Analysis

To test Hypothesis 1, the degree distribution for all systems is computed and, in accordance with Section 4.2, hierarchical links are eliminated from the analysis. To test Hypothesis 2, we compute the degree distribution for all systems excluding hierarchical and within-module links. Between-module links are identified using the approach outlined in Section 4.2: links are between-module if their source node and target node do not share the same class in their hierarchical ancestry.

All nodes that have a degree of zero (such as those whose sole dependency is through hierarchical relationships) are removed from the analysis. The resulting distributions are plotted using a log-log scale; the full set is available elsewhere.³

5.1 Overall Connectivity

For the purposes of discussion, three example plots are chosen based on system size (total node count). The example systems are *derby-10.1.1.0*, *jung-1.7.6*, and *picocontainer-1.3*, which represent the largest, median, and smallest systems, respectively. Figure 6 shows these distributions on a single plot. The similarity in shape is striking: we observe a positive slope between the first two data points,

³ <http://hdl.handle.net/10289/5307>

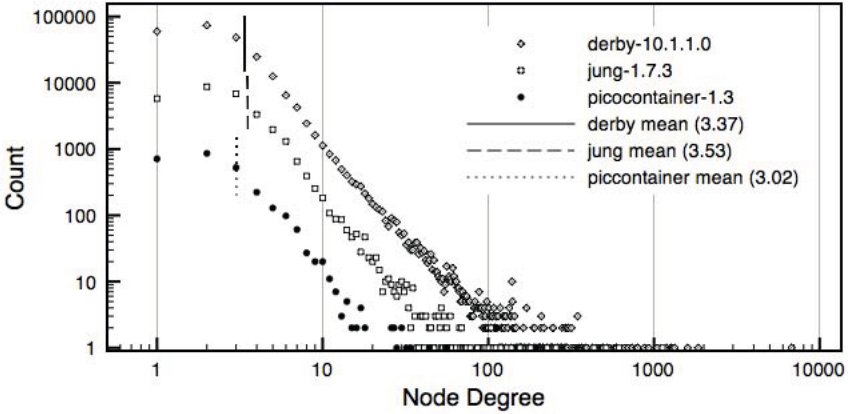


Fig. 6. Distribution of overall connectivity for the example systems

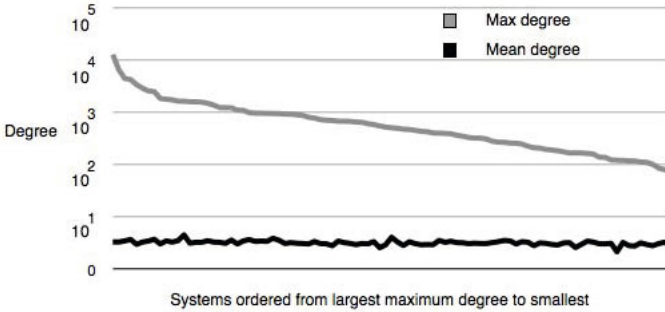


Fig. 7. Mean degree versus maximum degree

followed by a linear negative trend. Note that each of the distributions is noisy at the right end of the distribution, which is expected because they are produced from discrete data points and naturally have fewer points exhibiting high values.

All the plots exhibit characteristics of heavy-tailed distributions. They are left skewed and have a total range that is at least an order of magnitude larger than the mean. The mean degree for each example system is also shown on Figure 6. To demonstrate this for all systems, the mean and maximum degrees of each system are computed and plotted with a logarithmically-scaled y-axis in Figure 7, and the systems are sorted by descending order of maximum degree. The mean degree over all systems remains relatively constant, while the maximum is roughly between 10 and 1000 times the mean for all systems.

Further observation of the distributions in Figure 6 reveals clear differentiation of the three systems except to the right where the distributions are noisy. This is consistent with the expectations of a power-law distribution (Equation 1). The probability of a node with a high degree decreases proportionally to the degree; therefore, given a fixed α , the number of nodes with higher degrees increases for

systems that have more nodes. Based on these observations, we conclude that Hypothesis 1 is satisfied: overall connectivity for source code entities follows a heavy-tailed distribution for all systems within the corpus.

5.2 Between-Module Connectivity

The between-module distributions for the three example systems are shown in Figure 8. The between-module distributions show the overall coupling present in each system.

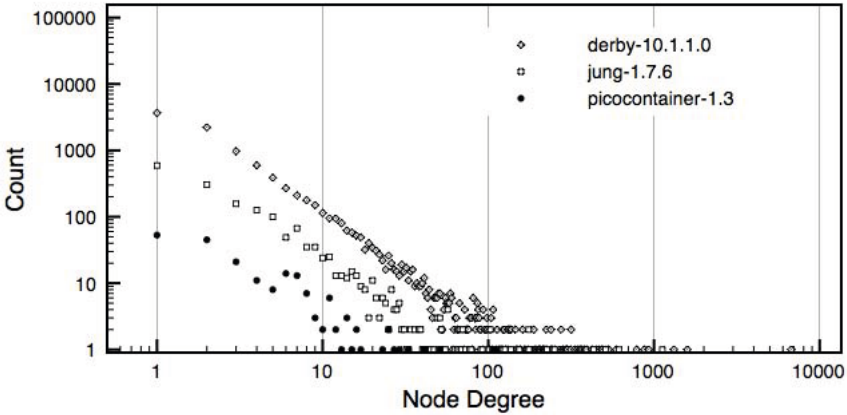


Fig. 8. Degree distributions for the sample systems (between-module only)

Figure 8 demonstrates that the between-module connectivity distributions are similar in shape to those computed to show overall connectivity (Figure 6). The between-module connectivity distributions are less well defined and this is due to the avoidance of between module interaction; less interaction equates with fewer data points, thereby producing noisier distributions. All the between-modules connectivity distributions have similar shape, including a heavy tail.

Figure 9 shows the overall and between-module connectivity distributions plotted together for each of the target systems. For each of the three systems we observe an overlap in the heavy tail. If the links for the nodes that exhibited high overall connectivity were primarily resolved within-module, then we would observe a migration of data points towards the left in the within-module distributions, which would therefore not exhibit a heavy tail. However, this migration is not observed. Instead, we observe heavy tails in both distributions, which overlap when we plot them on the same graph. This demonstrates that the nodes that appear in the heavy tail of the overall connectivity distributions are the same nodes that appear in the heavy tail of the between-module connectivity distributions, and are responsible for the presence of high-coupling.

In Figure 9, we observe that there is a difference in slope of the linear portions of the distributions. Because the overall connectivity distributions have more data points in the left side of the distribution, the slope in the overall connectivity distributions are steeper than the slope for the corresponding

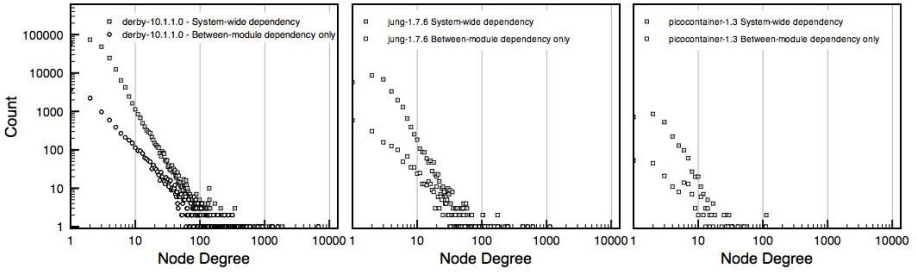


Fig. 9. Comparison of overall and between-module connectivity distributions for the example systems

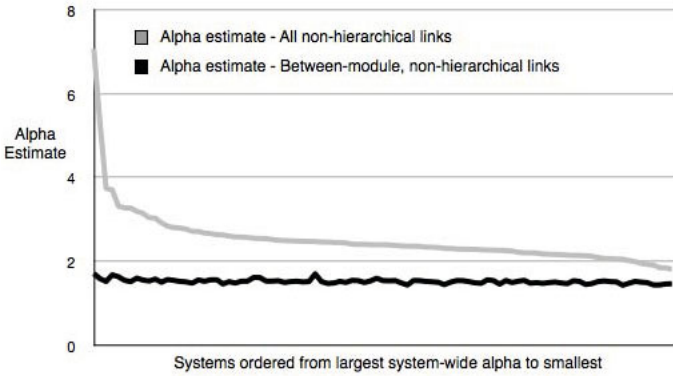


Fig. 10. Comparison of α estimates

within-module connectivity distributions. Using the process outlined in Section 2.1, we estimate α for all distributions. We use $d_{\min} = 1$ for all between-module distributions and $d_{\min} = 2$ for all overall connectivity distributions; data below the threshold are ignored. Comparison of estimated α between overall and within-module connectivity distributions for all systems (sorted in descending order by largest α estimate for overall connectivity) is shown on Figure 10. Estimated α for between-module connectivity distributions is lower than the overall connectivity distribution for the same system and this is true for all systems. Based on the above analysis, we conclude that Hypothesis 2 is satisfied.

6 Discussion

Here we discuss remaining issues and avenues for further research.

6.1 Threats to Validity

Internal validity. It is an important characteristic of these findings that scale-free structure was found to be ubiquitous in the data set. It is important because

we do not have *a priori* knowledge about the quality of design of the systems in the corpus and because of this, we have no ground truth from which to argue about the relationship between scale-free structure, high coupling, and the design of software systems. The results demonstrate, however, that high coupling is present in every system in our data set. If we are to accept high coupling as a definitive indicator of poor design, then we would have to conclude that all 97 systems under investigation suffer from poor design. This conclusion, however, seems implausible given the number of systems and the long modification and usage history and maturity of some of the systems. It is likely that at least some of the systems in the corpus are well designed despite the fact that they contain areas of high coupling. To be clear, we do not argue that all the systems in the corpus are well designed, but we argue against the notion that they are all poorly designed and that at least for the systems in the corpus, the presence of high coupling does not distinguish good design from poor. From these findings, we conclude that the presence of high coupling can be consistent with good design.

Construct validity. In Section 3.2, we presented a model of software evolution that is based on preferential attachment. Our model assumes hierarchical structure imposed as a constraint and utilizes a preferential probability function based on node functionality, module structure, and scoping rules. We are quick to point out that our finding of scale-free structure at the source code level does not necessarily imply our model in action. Keller notes that the mere presence of a particular distribution does not imply particular underlying or generational process [19]. However, our model does show that preferential attachment can be modified to be consistent with the evolution of software systems, thereby providing the possibility that our findings may translate to other programming languages and paradigms. If scale-free structure is common at the overall connectivity level, then high coupling is difficult to avoid.

External validity. While this study demonstrates that the presence of high coupling can be consistent with good design practice, we caution against extrapolating the specific structures identified in the examined systems to all software systems. All the systems in this investigation were open source and written using the Java programming language. Based on the structure of our investigation, it may be that the programming paradigm or open source nature of the systems confound our results. Different programming paradigms may produce dependency structures that are different from those generated using an object-oriented paradigm. Similarly, open source software may introduce greater levels of dependency through the desire to appeal to a broad base of users. Although our investigation did not identify any systems that did not contain areas of high coupling, we cannot conclude that a system with such structure does not exist.

The differing functionality, size, maturity, and modification histories of the investigated systems supports some generalizability of these findings. None of the systems under investigation were immune to the hypothesized effects, thereby suggesting that the presence of scale-free structure is independent of these properties.

6.2 Near-Constant α for Between-Module Connectivity

From our model, we believe that α for between-module connectivity is obtained through balancing two opposing goals: manageable module sizes and low coupling. While we expected to see distributions that were not extremely left-skewed (as stated in Hypothesis 2), we were surprised at the constancy of α . This may be a sign of an optimal balance that developers are able to achieve. One must recall that this is an exponent, however, and so even small variations can have large effects on the actual data. Even so, the possibility that this is more than coincidence is intriguing, and demands further investigation.

6.3 Varieties of Coupling

It has been argued that not all types of coupling are the same. Indeed, Wheeldon and Counsell [37] studied 5 different class-coupling relationships and found them to be independent. However, they also concluded that all 5 relationships followed a power-law, which suggests that high-coupling exists across those types. One avenue of future work suggests expanding the analysis performed here to account for different types of coupling.

There are cases where the quality of design is reduced as a tradeoff to a more desirable goal, such as performance optimizations. It is possible that some of the coupling detected by our analysis is the result of performance optimization; however, we consider it unlikely that all 97 systems have been subjected to this kind of optimization.

7 Conclusion

We have long heard the maxim of “high cohesion/low coupling” as a basis for good design. Abstract models of evolvability demonstrate why modularization and minimization of between-module connectivity (coupling) are essential to building complex systems: change propagation that would otherwise destabilize an unconstrained network can be contained. We build from the preferential attachment model and standard ideas of modularity to theorize as to why highly-coupled nodes should be expected in real software systems. In our model, we propose a preferential probability function based on entity utility and we argue that the probability of attachment to utility-providing nodes is not uniform because nodes will provide functionality of differing utility.

Using classes to define modules, we studied connectivity in 97 open source software systems using a graph-based analytic framework. Regardless of maturity, size, modification history, and the size of the user community, all these systems exhibit a similar scale-free dependency structure in both the structure of overall connectivity and between-module connectivity (coupling). Our analysis also demonstrated a relationship between highly-connected source code entities and high coupling: entities that exhibited high connectivity were the same entities that participated in areas of high coupling, as these nodes made up the heavy

tail of both distributions. The links of highly connected source code entities were not generally resolved within-module, and our model indicates that this is due to practical limits on module sizes. From this, we conclude that scale-free structure in the source code network translates directly to high coupling.

Thus, we conclude that high coupling is impracticable to eliminate entirely from software design. The maxim of “high cohesion/low coupling” is interpreted by some to mean that all occurrences of high coupling necessarily represent poor design. In contrast, our findings suggest that some high coupling is necessary for good design.

Acknowledgments. We would like to thank Robert Akscyn for feedback on the research and paper, the members of the Laboratory for Software Modification Research at the University of Calgary (especially Rylan Cottrell) for their devoted assistance over the years, and the anonymous reviewers for their diligence and professionalism. This work was funded by a Discovery Grant and a Postgraduate Scholarship (PGS D) from the Natural Sciences and Engineering Research Council of Canada. High-performance computing was provided by the Symphony High-Performance Computing Cluster at the University of Waikato.

References

1. Albert, R., Jeong, H., Barabási, A.-L.: Diameter of the World Wide Web. *Nature* 401, 130–131 (1999)
2. Alexander, C.: *Notes on the Synthesis of Form*. Harvard University Press, Cambridge (1964)
3. Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. *Science* 286(5439), 509–512 (1999)
4. Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E.: Understanding the shape of Java software. In: *Proc. ACM Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pp. 397–412 (2006)
5. Briand, L.C., Daly, J.W., Wüst, J.K.: A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.* 25(1), 91–121 (1999)
6. Chen, T., Gu, Q., Wang, S., Chen, X., Chen, D.: Module-based large-scale software evolution based on complex networks. In: *Proc. IEEE Int. Conf. Comp. Info. Technol.*, pp. 798–803 (2008)
7. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* 20(6), 476–493 (1994)
8. Clauset, A., Shalizi, C.R., Newman, M.E.J.: Power-law distributions in empirical data. *SIAM Rev.* 51(4), 661–703 (2009)
9. Concas, G., Marchesi, M., Pinna, S., Serra, N.: Power-laws in a large object-oriented software system. *IEEE Trans. Softw. Eng.* 33(10), 687–708 (2007)
10. Dijkstra, E.W.: Structured programming. In: Buxton, J.N., Randell, B. (eds.) *Software Engineering Techniques*, pp. 84–87. NATO Scientific Affairs Division, Brussels (1970)
11. Erdős, P., Rényi, A.: On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.* 5, 17–61 (1960)

12. Gao, Y., Xu, G., Yang, Y., Niu, X., Guo, S.: Empirical analysis of software coupling networks in object-oriented software systems. In: Proc. IEEE Int. Conf. Softw. Eng. Service Sci, pp. 178–181 (2010)
13. Goh, K.-I., Oh, E., Jeong, H., Kahng, B., Kim, D.: Classification of scale-free networks. Proc. Nat. Acad. Sci. 99(20), 12583–12588 (2002)
14. Hatton, L.: Power-law distributions of component size in general software systems. IEEE Trans. Softw. Eng. 35(4), 566–572 (2009)
15. Hyland-Wood, D., Carrington, D., Kaplan, S.: Scale-free nature of Java software package, class and method collaboration graphs. Technical Report TR-MS1286, University of Maryland, College Park (2006)
16. Ichii, M., Matsushita, M., Inoue, K.: An exploration of power-law in use-relation of Java software systems. In: Proc. Australian Conf. Softw. Eng., pp. 422–431 (2008)
17. Jenkins, S., Kirk, S.R.: Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. Info. Sci. 177, 2587–2601 (2007)
18. Jing, L., Keqing, H., Yutao, M., Rong, P.: Scale free in software metrics. In: Proc. Int. Comp. Softw. Appl. Conf. (2006)
19. Keller, E.F.: Revisiting “scale-free” networks. BioEssays 27(10), 1060–1068 (2005)
20. Koenig, A.: Patterns and antipatterns. J. Obj.-Oriented Progr. 8(1), 46–48 (1995)
21. Lehman, M.M.: Laws of software evolution revisited. In: Montangero, C. (ed.) EHSPT 1996. LNCS, vol. 1149, Springer, Heidelberg (1996)
22. Li, D., Han, Y., Hu, J.: Complex network thinking in software engineering. In: Proc. Int. Conf. Comp. Sci. Softw. Eng., pp. 264–268 (2008)
23. Louridas, P., Spinellis, D., Vlachos, V.: Power laws in software. ACM Trans. Softw. Eng. Methodol. 18(1), 2/1–2/26 (2008)
24. Marchesi, M., Pinna, S., Serra, N., Tuveri, S.: Power laws in Smalltalk. In: Proc. Europ. Smalltalk User Group Joint Event (2004)
25. Mens, T., Lanza, M.: A graph-based metamodel for object-oriented software metrics. Electr. Notes Theoret. Comp. Sci. 72(2) (2002)
26. Myers, C.R.: Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68, 046116 (2003)
27. Newman, M., Barabási, A.-L., Watts, D.J.: The Structure and Dynamics of Networks. Princeton University Press, Princeton (2006)
28. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM 15(12), 1053–1058 (1972)
29. Parnas, D.L.: On the design and development of program families. IEEE Trans. Softw. Eng. 2(1), 1–9 (1976)
30. Potanin, A., Noble, J., Fream, M., Biddle, R.: Scale-free geometry in object-oriented programs. Commun. ACM 48(5), 99–103 (2005)
31. Simon, H.A.: The architecture of complexity. Proc. Amer. Phil. Soc. 106(6), 467–482 (1962)
32. Stevens, W.P., Myers, G.J., Constantine, L.L.: Structured design. IBM Syst. J. 13(2), 231–256 (1974)
33. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: The Qualitas Corpus: A curated collection of Java code for empirical studies. In: Proc. Asia-Pacific Softw. Eng. Conf. (2010)
34. Valverde, S., Ferrer Cancho, R., Solé, R.V.: Scale-free networks from optimal design. Europhys. Lett. 60(4), 512–517 (2002)

35. Valverde, S., Solé, R.V.: Logarithmic growth dynamics in software networks. *Europhys. Lett.* 72(5), 858–864 (2005)
36. Vasa, R., Lumpe, M., Branch, P., Nierstrasz, O.: Comparative analysis of evolving software systems using the Gini coefficient. In: *Proc. IEEE Int. Conf. Softw. Maint.*, pp. 179–188 (2009)
37. Wheeldon, R., Counsell, S.: Power law distributions in class relationships. In: *Proc. IEEE Int. Wkshp. Source Code Analys. Manipul.*, pp. 45–54 (2001)
38. Wilkie, F.G., Kitchenham, B.A.: Coupling measures and change ripples in C++ application software. *J. Syst. Softw.* 52(2–3), 157–164 (2000)

Expressiveness, Simplicity, and Users

Craig Chambers

Google

Abstract. I have worked on several different language design and optimizing compiler projects, and I am often surprised by which ideas turn out to be the most successful. Oftentimes it is the simplest ideas that seem to get the most traction in the larger research or user community and therefore have the greatest impact. Ideas I might consider the most sophisticated and advanced can be challenging to communicate, leading to less influence and adoption. This effect is particularly pronounced when seeking to gain adoption among actual users, as opposed to other researchers. In this talk I will discuss examples of the tradeoffs among sophistication, simplicity, and impact in my previous research work in academia and in my current work at Google.

CDDiff: Semantic Differencing for Class Diagrams

Shahar Maoz*, Jan Oliver Ringert**, and Bernhard Rumpe

Software Engineering
RWTH Aachen University, Germany
<http://www.se-rwth.de/>

Abstract. Class diagrams (CDs), which specify classes and the relationships between them, are widely used for modeling the structure of object-oriented systems. As models, programs, and systems evolve over time, during the development lifecycle and beyond it, effective change management is a major challenge in software development, which has attracted much research efforts in recent years.

In this paper we present *cddiff*, a semantic diff operator for CDs. Unlike most existing approaches to model comparison, which compare the concrete or the abstract syntax of two given diagrams and output a list of syntactical changes or edit operations, *cddiff* considers the *semantics* of the diagrams at hand and outputs a set of *diff witnesses*, each of which is an object model that is possible in the first CD and is not possible in the second. We motivate the use of *cddiff*, formally define it, and show how it is computed. The computation is based on a reduction to Alloy. The work is implemented in a prototype Eclipse plug-in. Examples show the unique contribution of our approach to the state-of-the-art in version comparison and evolution analysis.

1 Introduction

Class diagrams (CDs) are widely used for modeling the structure of object-oriented systems. The syntax of CDs includes classes and the various relationships between them (association, generalization, etc.). The semantics of CDs is given in terms of object models, consisting of sets of objects and the relationships between these objects. Specifically, we are interested in a variant of the standard UML2 CDs, which is rich and expressive, supporting generalizations (inheritance), interface implementation, abstract and singleton classes, class attributes, uni- and bi-directional associations with multiplicities, enumerations, aggregation, and composition.

As models, programs, and systems evolve over time, during the development lifecycle and beyond it, effective change management and controlled evolution are major challenges in software development, and thus have attracted much research

* S. Maoz acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research.

** J.O. Ringert is supported by the DFG GK/1298 AlgoSyn.

efforts in recent years (see, e.g., [15,10,14,17,22,26,34]). Fundamental building blocks for tracking the evolution of software artifacts are diff operators one can use to compare two versions of a program or a model. Most existing approaches to differencing concentrate on matching between model elements using different heuristics related to their names and structure and on finding and presenting differences at a concrete or abstract syntactic level. While showing some success, most of these approaches are also limited. Models that are syntactically very similar may induce very different semantics (in the sense of ‘meaning’ [12]), and vice versa, models that semantically describe the same system may have rather different syntactic representations. Thus, a list of syntactic differences, although accurate, correct, and complete, may not be able to reveal the real implications these differences may have on the correctness and potential use of the models involved. In other words, such a list, although easy to follow, understand, and manipulate (e.g., for merging), may not be able to expose and represent the semantic differences between two versions of a model, in terms of the bugs that were fixed or the features (and new bugs...) that were added.

In this paper we present *cddiff*, a semantic diff operator for CDs. Unlike existing differencing approaches, *cddiff* is a *semantic diff operator*. Rather than comparing the concrete or the abstract syntax of two given diagrams, and outputting a list of syntactical changes or edit operations, *cddiff* considers the semantics of the diagrams at hand and outputs a set of *diff witnesses*, each of which is an object model that is possible in the first CD and is not possible in the second. These object models provide concrete proofs for the meaning of the change that has been done between the two compared versions and for its effect on the use of the models at hand.

We specify CDs using the class diagrams of UML/P [29], a conceptually refined and simplified variant of UML designed for low-level design and implementation. Our semantics of CDs is based on [11] and is given in terms of sets of objects and relationships between these objects. An overview of the formal definition of the syntax and semantics of our CDs is given in Sect. 3.

Given two CDs, cd_1 and cd_2 , $cddiff(cd_1, cd_2)$ is roughly defined as the set of object models possible in the first CD and not possible in the second. As this set may be infinite, we are specifically interested in its bounded version, $cddiff_k(cd_1, cd_2)$, which only includes object models where the number of object instances is not greater than k . The formal definition of *cddiff* is given in Sect. 4.

To compute *cddiff* we use Alloy [13]. Alloy is a textual modeling language based on relational first-order logic. A short overview of Alloy is given in Sect. 3.2. To employ Alloy for our needs, we have defined a transformation that takes two CDs and generates a single Alloy module. The module includes predicates specifying each of the CDs, $cd1$ and $cd2$, and a diff predicate reading $Cd1NotCd2$, specifying the existence of a satisfying assignment for the predicate $cd1$ (for us, representing an instance of cd_1), which is not a satisfying assignment for the predicate $cd2$ (representing an instance of cd_2). Analyzing this predicate with a user-specified scope k produces elements of $cddiff_k(cd_1, cd_2)$, if any, as required.

Our transformation is very different from ones suggested in other works that use Alloy to analyze CDs (see, e.g., [4,21]). First, we take two CDs as input, and output one Alloy module. Second, to support a comparison in the presence of generalizations and associations, we must use a non-shallow embedding, that is we have to encode all the relationships between the signatures in generated predicates ourselves. In particular, we cannot use Alloy’s `extends` keyword to model inheritance and cannot use Alloy’s fields to model class attributes, because for the shared signatures, a class’s inheritance relation and set of attributes may be different between the two CDs. Thus, these need to be modeled as predicates, different ones for each CD, outside the signatures themselves. The transformation is described in Sect. 4.2.

In addition to finding concrete diff witnesses (if any exist), which demonstrate the meaning of the changes that were made between one version and another, *cddiff* can be used to compare two CDs and decide whether one CD semantics includes the other CD semantics (the latter is a refinement of the former), are they semantically equivalent, or are they semantically incomparable (each allows instantiations that the other does not allow). When applied to the version history of a certain CD, which can be retrieved from a version repository, such an analysis provides a semantic insight into the evolution of this CD, which is not available in existing syntactic approaches.

We have implemented *cddiff* and integrated it into a prototype Eclipse plug-in. The plug-in allows the engineer to compare two selected CDs and to browse the diff witnesses found, if any. Indeed, all examples shown in this paper have been computed by our plug-in. We describe the plug-in’s implementation, main features, and performance results in Sect. 5.

Following the evaluation in Sect. 5, we define and implement two important extensions of the basic *cddiff* technique. The first extension deals with filtering the diff witnesses found, so that ‘uninteresting witnesses’ are filtered out, and a more succinct yet informative set of witnesses is provided to the engineer. The second extension deals with the use of abstraction in the comparison. The extensions are described in Sect. 6.

Model and program differencing, in the context of software evolution, has attracted much research efforts in recent years (see [1,5,10,14,17,22,26,34]). In contrast to our work, however, most studies in this area present syntactic differencing, at either the concrete or the abstract syntax level. We discuss related work in Sect. 8.

It is important not to confuse differencing with merging. Merging is a very important problem, dealing with reconciling the differences between two models that have evolved independently from a single source model, by different developers, and now need to be merged back into a single model (see, e.g., [3,10,17,23,25]). Differencing, however, is the problem of identifying the differences between two versions, for example, an old version and a new one, so as to better understand the course of a model evolution during some step of its development. Thus, diff witnesses are not conflicts that need to be reconciled. Rather, they are proofs

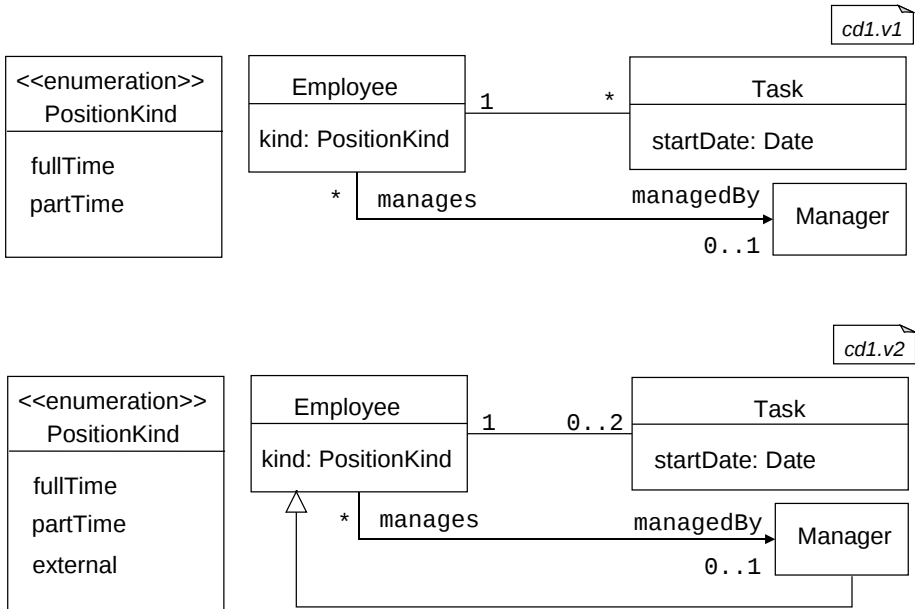


Fig. 1. *cd1.v1* and its revised version *cd1.v2*

of features that were added or bugs that have been fixed from one version to another along the history of the design and development process.

The next section presents motivating examples demonstrating the unique features of our work. Sect. 3 provides preliminary definitions of the CD language syntax and semantics as used in our work. Sect. 4 introduces *cddiff* and the technique to compute it. Sect. 5 presents the prototype implementation and related applications. Sect. 6 describes the filtering and abstraction extensions. Sect. 7 presents a discussion of advanced topics and future work directions, Sect. 8 considers related work, and Sect. 9 concludes.

2 Examples

We start off with motivating examples for semantic differencing of CDs. The examples are presented semi-formally. Formal definitions appear in Sect. 4.

2.1 Example I

Consider *cd1.v1* of Fig. 1, describing a first version of a model for (part of) a company structure with employees, managers, and tasks. A design review with a domain expert has revealed three bugs in this model: first, employees should not be assigned more than two tasks; second, managers are also employees, and they can handle tasks too; and third, there is another kind of position, namely an external position.

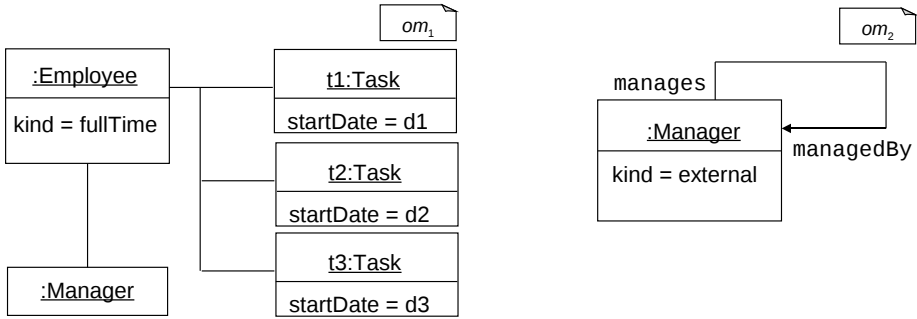


Fig. 2. Example object models representing semantic differences between the old class diagram $cd1.v1$ and its revised version $cd1.v2$

Following this design review, the engineers created a new version $cd1.v2$, shown in the same figure. The two versions share the same set of named elements but they are not identical. Syntactically, the engineers added an inheritance relation between **Manager** and **Employee**, set the multiplicity on the association between **Employee** and **Task** to 0..2, and added the external position kind. What are the semantic consequences of these differences?

Using the operator $cddiff$ we can answer this question. $cddiff(cd1.v1, cd1.v2)$ outputs om_1 , shown in Fig. 2, as a diff witness that is in the semantics of $cd1.v1$ and not in the semantics of $cd1.v2$; thus, it demonstrates (though does not prove) that the bug of having more than two tasks per employee was fixed. In addition, $cddiff(cd1.v2, cd1.v1)$ outputs om_2 , shown in Fig. 2 too. om_2 is a diff witness that is in the semantics of the new version $cd1.v2$ and not in the semantics of the old version $cd1.v1$. Thus, the engineers should perhaps check with the domain expert whether the model should indeed allow managers to manage themselves and hold an external kind of position.

2.2 Example II

The two class diagrams $cd3.v1$ and $cd3.v2$, shown in Fig. 3, provide alternative descriptions for the relation between **Department** and **Employee** in the company. Again the two diagrams share the same set of named elements but the diagrams are not identical. First, **Department** is a singleton only in $cd3.v1$. Second, only in $cd3.v1$ the relation between **Department** and **Employee** is a Whole/Part composition relation. What are the semantic consequences of the differences between the two versions of $cd3$?

Fig. 3 includes two objects models. In om_3 there are two departments with no employees. In om_4 there is a single employee and no departments. It is easy to see that both object models are in the semantics of $cd3.v2$ but not in the semantics of $cd3.v1$. We formally write it as $\{om_3, om_4\} \subseteq cddiff(cd3.v2, cd3.v1)$. In addition, we can see that $cd3.v2$ is a refinement of $cd3.v1$, since all object models in the semantics of $cd3.v1$ are also in the semantics of $cd3.v2$ (that is, $cddiff(cd3.v1, cd3.v2) = \emptyset$). Again, the two diff witnesses (in one direction) can

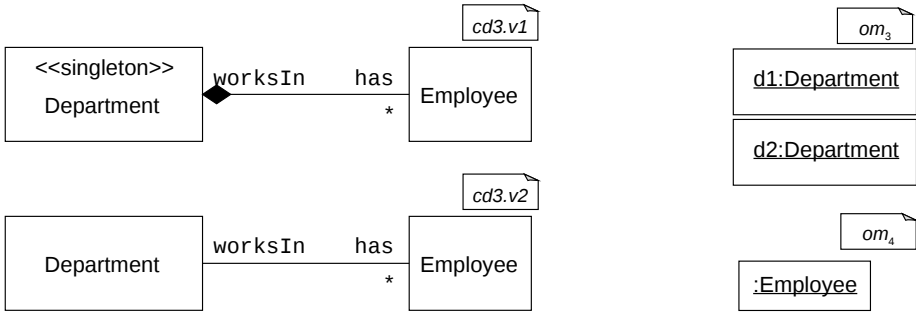


Fig. 3. *cd3.v1* and its revised version *cd3.v2*, with example object models representing the semantic differences between them. Both object models are in the semantics of *cd3.v2* and not in the semantics of *cd3.v1*.

be computed and the refinement relation (in the other direction) can be proved (in a bounded scope) by our operator.

2.3 Example III

Finally, *cd5.v1* of Fig. 4 is another class diagram from this model of company structure. In the process of model quality improvement, an engineer has suggested to refactor it by introducing an abstract class **Person**, replacing the association between **Employee** and **Address** by an association between **Person** and **Address**, and redefining **Employee** to be a subclass of **Person**. The resulting suggested CD is *cd5.v2*.

Using *cddiff* we are able to prove (in a bounded scope) that despite the syntactic differences, the semantics of the new version is equivalent to the semantics of the old one, formally written $cddiff(cd5.v1, cd5.v2) = cddiff(cd5.v2, cd5.v1) = \emptyset$. The refactoring is correct and the new suggested version can be committed.

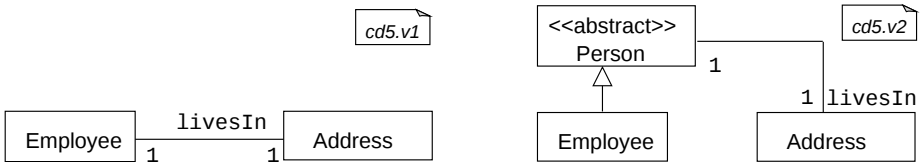


Fig. 4. *cd5.v1* and its revised version *cd5.v2*. The two versions have equal semantics

3 Preliminaries

We give a short overview of the CD language used in our work and of Alloy, the tool we use for the computation of *cddiff*.

3.1 Class Diagrams Language

As a concrete CD language we use the class diagrams of UML/P [29], a conceptually refined and simplified variant of UML designed for low-level design and implementation. Our semantics of CDs is based on [11] and is given in terms of sets of objects and relationships between these objects. More formally, the semantics is defined using three parts: a precise definition of the syntactic domain, i.e., the syntax of the modeling language CD and its context conditions (we use MontiCore [16,24] for this); a semantic domain - for us, a subset of the System Model (see [7,8]) OM, consisting of all finite object models; and a mapping $sem : CD \rightarrow \mathcal{P}(OM)$, which relates each syntactically well-formed CD to a set of constructs in the semantic domain OM. For a thorough and formal account of the semantics see [8].

Note that we use a *complete* interpretation for CDs (see [29] ch. 3.4). This roughly means that ‘whatever is not in the CD, should indeed not be present in the object model’. In particular, we assume that the list of attributes of each class is complete, e.g., an `employee` object with an `id` and a `salary` is not considered as part of the semantics of an `Employee` class with an `id` only.

The CD language constructs we support include generalization (inheritance), interface implementation, abstract and singleton classes, class attributes, uni- and bi-directional associations with multiplicities, enumerations, aggregation, and composition.

3.2 A Brief Overview of Alloy

Alloy [2,13] is a textual modeling language based on relational first-order logic. An Alloy module consists of a number of signature declarations, fields, facts and predicates. The basic entities in Alloy are atoms. Each signature denotes a set of atoms. Each field belongs to a signature and represents a relation between two or more signatures. Such relations are interpreted as sets of tuples of atoms. Facts are statements that define constraints on the elements of the module. Predicates are parametrized constraints, which can be included in other predicates or facts.

Alloy Analyzer is a fully automated constraint solver for Alloy modules. The analysis is achieved by an automated translation of the module into a Boolean expression, which is analyzed by SAT solvers embedded within the Analyzer. The analysis is based on an exhaustive search for instances of the module. The search space is bounded by a user-specified scope, a positive integer that limits the number of atoms for each signature in an instance of the system that the solver analyzes.

The Analyzer can check for the validity of user-specified assertions. If an instance that violates the assertion is found within the scope, the assertion is not valid. If no instance is found, the assertion might be invalid in a larger scope. Used in the opposite way, one can look for instances of user-specified predicates. If the predicate is satisfiable within the given scope, the Analyzer will find an instance that proves it. If not, the predicate may be satisfiable in a larger scope. We discuss the advantages and limitations of using Alloy for our problem in Sect. 7. A thorough account of Alloy can be found in [13].

4 CDDiff

4.1 Definitions

We define a diff operator $cddiff : CD \times CD \rightarrow \mathcal{P}(OM)$, which maps two CDs, cd_1 and cd_2 , to the (possibly infinite) set of all object models that are in the semantics of cd_1 and are not in the semantics of cd_2 . Formally:

Definition 1. $cddiff(cd_1, cd_2) = \{om \in OM \mid om \in sem(cd_1) \wedge om \notin sem(cd_2)\}$.

Note that $cddiff$ is not symmetric. In addition, by definition, $\forall cd_1, cd_2 \in CD$, $cddiff(cd_1, cd_1) = \emptyset$ (the empty set, not the empty object model) and $cddiff(cd_1, cd_2) \cap cddiff(cd_2, cd_1) = \emptyset$, as expected. The members of the set $cddiff$ are called *diff witnesses*.

The set-theoretic definition of $cddiff$, as given above, is however not constructive, and may yield an infinite set. As a pragmatic solution, we approximate it by defining (a family of) bounded diff operators that we are able to compute. Thus, we use a bound k , which limits the total number of objects in the diff witnesses we are looking for. Formally:

Definition 2. $\forall k \geq 0$, $cddiff_k(cd_1, cd_2) = \{om \mid om \in cddiff(cd_1, cd_2) \wedge |om| \leq k\}$, where $|om|$ is the total number of objects in om .

4.2 Computing $cddiff_k$: Overview

To compute $cddiff_k$ we use Alloy. To employ Alloy for our needs, we have defined a transformation that takes two CDs and generates a single Alloy module. The module includes predicates specifying each of the CDs, `cd1` and `cd2`, and a diff predicate reading `Cd1NotCd2`, specifying the existence of a satisfying assignment for the predicate `cd1` (for us, representing an instance of cd_1), which is not a satisfying assignment for the predicate `cd2` (representing an instance of cd_2). Analyzing this predicate with a user-specified scope k produces elements of $cddiff_k(cd_1, cd_2)$, if any, as required.

Our transformation is very different from ones suggested in other works that use Alloy to analyze CDs (see, e.g., [421]). First, we take two CDs as input, and output a single Alloy module. Second, to support a comparison in the presence of generalizations and associations, we must use a non-shallow embedding, that is we have to encode all the relationships between the signatures in generated predicates ourselves. In particular, we cannot use Alloy’s `extends` keyword to model inheritance and cannot use Alloy’s fields to model class attributes, because for the shared signatures, a class’s inheritance relation and set of attributes may be different between the two CDs. Thus, these need to be modeled as predicates, different ones for each CD, outside the signatures themselves.

It is important to note that a naive approach that would transform each of the two CDs separately into a corresponding Alloy module and then compare the instances found by the analyzer for each CD, would have been incomplete and hopelessly inefficient. Such an approach requires the complete computation of

the two sets of instances before the comparison could be done. As Alloy generates instances one-by-one, with no guarantee about their order, this could not work in practice. Thus, our approach, of taking the two input CDs and constructing a single Alloy module whose all instances, if any, are diff witnesses, is indeed required. In other words, instead of computing the differences, if any, ourselves, we create an Alloy module whose instances are the differences we are looking for, and let the SAT solver do the hard work for us.

Below we show only selected excerpts from the generated Alloy module corresponding to the two CDs from the example in Fig. 1 (a complete definition of the translation, which shows how each CD construct is handled, is given in supporting materials available from [31]).

4.3 Computing $cddiff_k$: The Generated Alloy Module

We start off with a generic part, which is common to all our generated modules.

```

1 // Names of fields/associations in classes of the model
2 abstract sig FName {}
3
4 // Parent of all classes relating fields and values
5 abstract sig Obj { get: FName -> {Obj + Val + EnumVal}}
6
7 // Values of fields
8 abstract sig Val {}
9
10 fact values {
11   // No values can exist on their own
12   all v: Val | some f: FName | v in Obj.get[f] }
13
14 //Names of enum values in enums of the model
15 abstract sig EnumVal {}
16
17 fact enums {
18   //no enum values can exist on their own
19   all v: EnumVal | some f: FName | v in Obj.get[f] }

```

Listing 1.1. FName, Obj, Val, and EnumVal signatures

List. 1.1 shows the abstract signature FName used to represent association role names and attribute names for all classes in the module. The abstract Obj signature is the parent of all classes in the module, and its get Alloy field relates it and an FName to instances of Obj, Val, and EnumVal. List. 1.1 also shows the abstract signature Val, which we use to represent all predefined types (i.e., primitive types and other types that are not defined as classes in the CDs). Values of enumeration types are represented using signature EnumVal. Enumeration values as well as primitive values should only appear in an instance if referenced by any object (see predicates in lines 10-12 and lines 17-19).

```

1 pred ObjAttrib[objs:set Obj,
2     fName:one FName, fType:set {Obj + Val + EnumVal}] {
3     objs.get[fName] in fType
4     all o: objs | one o.get[fName] }
5
6 pred ObjNoFName[objs:set Obj, fName:set FName] {
7     no objs.get[fName] }

```

Listing 1.2. Predicates for objects and their fields

List. [1.2](#) shows some of the generated predicates responsible for specifying the relation between objects and fields: `ObjAttrib` limits `objs.get[fName]` to the correct field's type and ensures that there is exactly one atom related to the field name (by the `get` relation); `ObjNoFName` is used to ensure classes do not have field names other than the ones stated in the CD.

```

1 pred ObjUAttrib[objs:set Obj,
2     fName:one FName, fType:set Obj, up: Int] {
3     objs.get[fName] in fType
4     all o: objs | (#o.get[fName] =< up) }
5
6 pred Composition[left:set Obj,
7     lFName:one FName, right:set Obj] {
8     all l1, l2: left |
9     (# {l1.get[lFName] & l2.get[lFName]} > 0) => l1=l2
10    all r: right | # {l: left | r in l.get[lFName]} = 1 }

```

Listing 1.3. Predicates for multiplicities and Whole/Part compositions

List. [1.3](#) shows some of the generated predicates responsible to specify multiplicities and Whole/Part compositions. The first predicate provides an upper bound for the number of objects in the set represented by the `get` relation for a specified role name. The second predicate is used to constrain a composition relation between classes. Its first statement (lines 8-9) ensures that no two wholes (on the 'left') own the same part (on the 'right'). The second statement (line 10) ensures that a part (on the 'right') belongs to exactly one whole (on the 'left').

```

1 // Predicate for diff
2 pred Cd1NotCd2 { cd1 not cd2}
3
4 // Command for diff
5 run Cd1NotCd2 for 5

```

Listing 1.4. The diff predicate and the related run command

List. [1.4](#) shows the simple predicate `Cd1NotCd2` representing the diff. An Alloy instance that satisfies the generated predicate `cd1` and does not satisfy the generated predicate `cd2` is in the set $cddiff_k(cd_1, cd_2)$. The value for the scope k of the `run` command (line 5) is part of the input of our transformation.

All the above are generic, that is, they are common to all generated modules, independent of the input CDs at hand. We now move to the parts that are specific to the two input CDs.

All class names and field names from the two CDs are shown in List. [1.5](#) as Alloy signatures and are stripped from their inheritance relations, attributes, associations etc. Note the `type_Date` signature in line 6, which extends `Val` (see List. [1.1](#)). Concrete enumeration values from both class diagrams are declared in lines 9-10.

```

1 // Concrete names of fields in cd1 and cd2
2 one sig startDate, mngBy, worksOn, mng,
3     doneBy, kind extends FName {}
4
5 // Concrete value types in model cd1 and cd2
6 lone sig type_Date extends Val {}
7
8 // Concrete enum values
9 lone sig enum_PosKnd_external, enum_PosKnd_fullTime,
10     enum_PosKnd_partTime extends EnumVal {}
11
12 // Actual classes in the model
13 sig Tsk, Emp, Mgr extends Obj {}

```

Listing 1.5. The common signatures

Next, we define a set of functions and a predicate for each CD individually. We show here only the ones for `cd1.v2`, a CD that we presented in Fig. [1](#) (in the generated Alloy code that we show below, this CD appears as `cd2`).

First, subtype functions, shown in List. [1.6](#) (top), which specify subtype relations between the relevant signatures specific for this CD. Note how function `EmpSubsCD2` denotes that in `cd1.v2` employees are either of type `Emp` or their subtype `Mgr`. The possible values of enumeration `PosKnd` in `cd1.v2` are defined by function `PosKndEnumCD2`.

Second and finally, the predicate `cd2`, specifying the properties of the CD `cd1.v2`, is shown in List. [1.6](#). Note the use of the generic predicates defined earlier, in particular, the use of the parametrized predicate `ObjNoFName` (defined in List. [1.2](#)); e.g., line 15 specifies that a `Tsk` has no other field names but `doneBy` and `startDate`. Also, note the use of the parametrized predicate `ObjLUAttrib` (defined using the predicate shown in List. [1.3](#)); e.g., line 27 specifies that all instances of `Emp` (including subtypes, see the function `EmpSubsCD2` defined in List. [1.6](#)), work on at most 2 tasks.

As an optional optimization, the transformation identifies and ignores syntactically equal attributes of same-name classes and common enumeration values

```

1 // Types wrapping subtypes
2 fun MgrSubsCD2: set Obj { Mgr}
3 fun TskSubsCD2: set Obj { Tsk}
4 fun EmpSubsCD2: set Obj { Mgr + Emp}
5
6 // Enums
7 fun PosKndEnumCD2: set EnumVal { enum_PosKnd_external +
8   enum_PosKnd_fullTime + enum_PosKnd_partTime }
9
10 // Values and relations in cd2
11 pred cd2 {
12
13   // Definition of class Tsk
14   ObjAttrib[Tsk, startDate, type_Date]
15   ObjNoFName[Tsk, FName - doneBy - startDate]
16
17   // Definition of class Emp
18   ObjAttrib[Emp, kind, PosKndEnumCD2]
19   ObjNoFName[Emp, FName - kind - mngBy - worksOn]
20
21   // Definition of class Mgr
22   ObjAttrib[Mgr, kind, PosKndEnumCD2]
23   ObjNoFName[Mgr, FName - kind - mngBy - worksOn]
24
25   // Associations
26   BidiAssoc[EmpSubsCD2, worksOn, TskSubsCD2, doneBy]
27   ObjLUAttrib[EmpSubsCD2, worksOn, TskSubsCD2, 0, 2]
28   ObjLUAttrib[TskSubsCD2, doneBy, EmpSubsCD2, 1, 1]
29
30   ObjLUAttrib[EmpSubsCD2, mngBy, MgrSubsCD2, 0, 1]
31   ObjL[MgrSubsCD2, mngBy, EmpSubsCD2, 0] }

```

Listing 1.6. Subtyping functions and the predicate for *cd1.v2*

between the two CDs. By definition, such attributes and enumerations will not be a necessary part of any diff witness and thus they can be ignored. Note that this is done on the flattened model, that is, while considering also inherited attributes. In addition to faster performance, this has the very important effect of reducing the size of the problem for Alloy, and hence, let us increase the maximum number of instances – Alloy’s scope – in finding a witness, while keeping the size of the SAT problem small, and thus better cope with the bounded analysis limitation. In particular, in the presence of large CDs, it allows us to find differences that we were unable to find otherwise.

5 Implementation and Evaluation

We have implemented *cddiff* and integrated it into a prototype Eclipse plugin. The input for the implementation are UML/P CDs, textually specified using

MontiCore grammar and generated Eclipse editor [16,24]. The plug-in transforms the input CDs into an Alloy module and uses Alloy’s APIs to analyze it and to produce diff witnesses. Witnesses are presented to the engineer using MontiCore object diagrams. The complete analysis cycle, from parsing the two selected CDs, to building the input for Alloy, to running Alloy, and to translating the Alloy instances that were found, if any, back to MontiCore object diagrams, is fully automated.

5.1 Browsing Diff Witnesses

The plug-in allows the engineer to compare two selected CDs, and to browse the diff witnesses found, if any. Fig. 5 shows an example screen capture, where the engineer has selected to compare *cd1.v1* (left) and *cd1.v2* (right), which we presented in Sect. 2, and is currently browsing one of the two diff witnesses that were found. This witness is an object diagram with a full-time employee handling three tasks.

Clicking **Compute** computes the diff witnesses and shows a message telling the engineer if any were found. The diff witness is textually displayed as an object diagram in the central lower pane. The **Next** and **Previous** buttons browse for the next and previous diff witnesses. The **Switch Left/Right** button switches the order of comparison. The **Settings** button opens a dialog that allows the engineer to set values for several parameters, such as the scope that Alloy should use in the computation and the activation of various filters and abstractions (see Sect. 6).

5.2 High-Level Evolution Analysis

Another application enabled by the plug-in is high-level evolution analysis. The plug-in supports a **compare** command: given two CDs, cd_1 and cd_2 , and a scope k , the command checks whether one CD is a refinement of the other, are the two CDs semantically equivalent, or are they semantically incomparable (each allows object models the other does not allow). Formally, $compare(cd_1, cd_2, k)$ returns one of four answers:

$$\begin{aligned}
 <_k & \quad \text{if } cddiff_k(cd_1, cd_2) = \emptyset \text{ and } cddiff_k(cd_2, cd_1) \neq \emptyset \\
 >_k & \quad \text{if } cddiff_k(cd_1, cd_2) \neq \emptyset \text{ and } cddiff_k(cd_2, cd_1) = \emptyset \\
 \equiv_k & \quad \text{if } cddiff_k(cd_1, cd_2) = \emptyset \text{ and } cddiff_k(cd_2, cd_1) = \emptyset \\
 <>_k & \quad \text{if } cddiff_k(cd_1, cd_2) \neq \emptyset \text{ and } cddiff_k(cd_2, cd_1) \neq \emptyset
 \end{aligned}$$

The subscript k denotes the scope used in the computation.

Given a reference to a series of historical versions of a CD, as can be retrieved from the CD’s entry in a revision repository (such as SVN, CVS etc.), the plug-in can use the **compare** command to compute a high-level analysis of the evolution of the CD: which new versions have introduced new possible implementations relative to their predecessors, which new versions have eliminated possible implementations relative to their predecessors, and which new versions included only syntactical changes that have not changed the semantics of the CD.

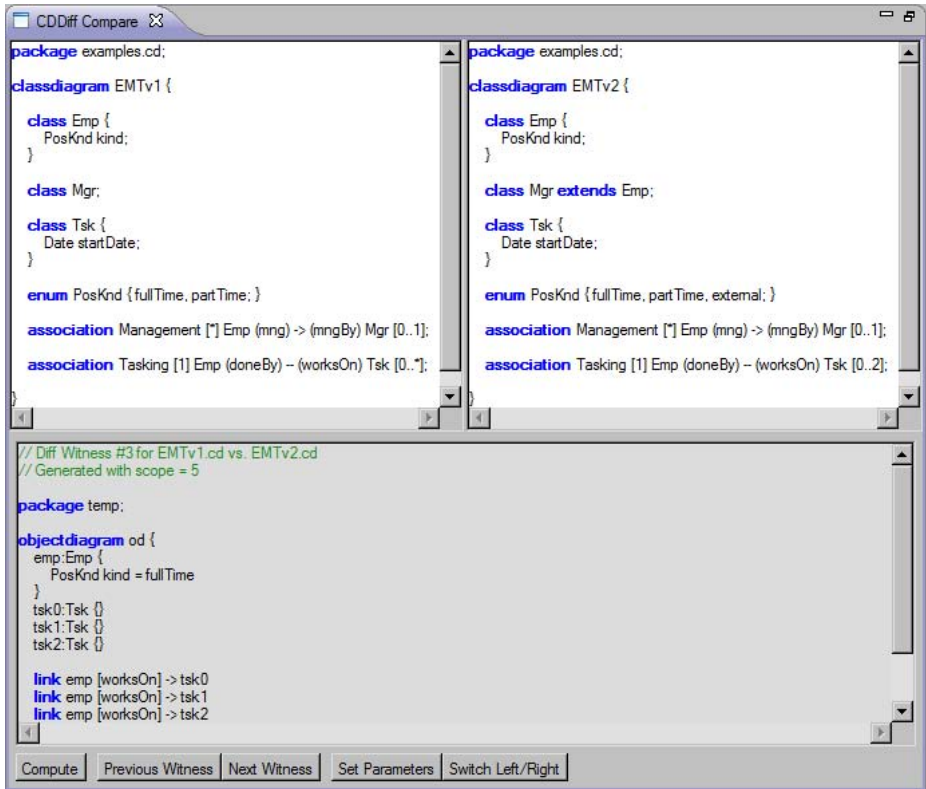


Fig. 5. A screen capture from Eclipse, showing a view from the prototype plug-in for *cdiff*. Two class diagrams that were selected by the user, corresponding to *cd1.v1* and *cd1.v2* of Fig. 11, are shown at the upper part of the screen. A generated diff witness, consisting of an object model that includes a full-time employee with three tasks, is displayed at the lower part of the screen.

For example, applying this evolution analysis to the examples presented in Sect. 2 with, e.g., a scope of 5, reveals: $compare(cd1.v1, cd1.v2, 5) = \langle \rangle_5$, $compare(cd3.v1, cd3.v2, 5) = \langle \rangle_5$, and $compare(cd5.v1, cd5.v2, 5) = \equiv_5$. Thus, it shows (within the selected scope), that *cd1.v1* and *cd1.v2* are incomparable (each allows object models that are not allowed by the other), that *cd3.v1* is a refinement of *cd3.v2* (the latter allows all the object models that are allowed by the former, and some more), and that *cd5.v1* and *cd5.v2* have equivalent semantics (one is a correct refactoring of the other).

5.3 Performance

We report the performance of the plug-in in generating diff witnesses. Experiments were done using Alloy version 4.1.10 with SAT4J [30], on a laptop computer, Intel Dual Core CPU, 2.8 GHz, with 4 GB RAM, running Windows Vista.

Table 1 shows results from computing diff witnesses for the three examples presented in Sect. 2 using different scopes. Each example is reported twice, computing the differences in both directions. The column titled **Vars / primary vars / clauses** reports on the SAT formula created by Alloy. The column titled **Alloy time** reports the time it took Alloy to find the first diff witness (building the formula + finding the instance). The column titled **# Witnesses** reports on the total number of witnesses found by the plug-in (we compute only the first 20 witnesses). The rightmost column reports the total time it took for the plug-in to compute all (up to 20) witnesses. All timing data is reported in milliseconds.

Table 2 shows the results from computing diff witnesses for several versions of CDs from a library example (The CDs of the library example are available for download as supporting materials in [31]). The CDs in this example include 11 classes, 4 enumerations (with average of 4 values each), 7 associations (with most multiplicities * or 1..*), an average of 4 attributes per class (some classes have 6 attributes), and an inheritance hierarchy of depth 3. The columns in Table 2 are the same as the ones in Table 1.

On the one hand, the performance results show that for relatively small models, computing diff witnesses using our approach runs very fast or at least in reasonable times. On the other hand, the results show that for large models, or ones that require a high scope, performance may not scale well, as doubling the scope typically causes a performance slowdown of a factor of 4 or more. Given these results, in the future, we plan to develop heuristics to improve the scalability of *cddiff*, using, e.g., abstraction / refinement techniques, decomposition for early detection of independent sub models, etc. See the short discussions in the next section.

Finally, as can be seen from the table, in all cases where witnesses exist, the plug-in has found 20 witnesses (and could have perhaps found more if we would have continued to look for more witnesses). This points to a limitation in *cddiff*, where despite the symmetry breaking heuristics employed by Alloy, many of the witnesses found are rather similar and thus not interesting. To address this limitation, we have defined and implemented a filtering mechanism. We discuss this in Sect. 6.1.

6 Extensions: Filtering and Abstraction

6.1 Filtering Diff Witnesses

One limitation of *cddiff* and its computation through Alloy as presented in previous sections, is related to the usefulness of the set of witnesses that we find. In some cases, the automatically generated set contains many very similar and thus possibly uninteresting witnesses. This is true despite the symmetry reduction heuristics employed by Alloy. For example, assuming a difference in multiplicities of * and 0..*m* between employees and tasks, all object models with one or more employees, where at least one employee has more than *m* tasks are diff witnesses. Indeed, all such witnesses (up to the specified scope) may be

Table 1. Results from computing diff witnesses for the three examples presented in Sect. 2, using different scopes. Each example is reported twice, computing the differences in both directions. The column titled **Vars / p. vars / clauses** reports on the SAT formula created by Alloy. The column titled **Alloy time** reports the time it took Alloy to find the first diff witness (building the formula + finding the instance). The column titled **# Wit.** reports on the total number of witnesses found by the plug-in (we compute up to 20 witnesses). The rightmost column reports the total time it took for the plug-in to compute all (up to 20) witnesses. All timing data is reported in milliseconds.

Name	Scope	Vars/p. vars/clauses	Alloy time (ms)	# Wit.	Plug-in time (ms)
Ex. 1	5	4079 / 234 / 9106	54 + 11	20	281
Ex. 1 rev.	5	4079 / 234 / 9092	44 + 7	20	212
Ex. 1	10	13664 / 704 / 33386	265 + 29	20	634
Ex. 1 rev.	10	13664 / 704 / 33357	253 + 20	20	603
Ex. 1	20	49834 / 2394 / 126446	1740 + 156	20	3472
Ex. 1 rev.	20	49834 / 2394 / 126387	1786 + 112	20	2970
Ex. 2	5	883 / 72 / 2014	8 + 2	0	11
Ex. 2 rev.	5	883 / 72 / 2014	8 + 1	20	76
Ex. 2	10	3613 / 242 / 9244	40 + 2	0	44
Ex. 2 rev.	10	3613 / 242 / 9244	40 + 5	20	164
Ex. 2	20	14713 / 882 / 39484	337 + 10	0	348
Ex. 2 rev.	20	14713 / 882 / 39484	347 + 18	20	814
Ex. 3	5	1165 / 77 / 2665	10 + 3	0	14
Ex. 3 rev.	5	1165 / 77 / 2665	10 + 3	0	14
Ex. 3	10	4455 / 252 / 11020	56 + 28	0	84
Ex. 3 rev.	10	4455 / 252 / 11020	49 + 20	0	70
Ex. 3	20	17575 / 902 / 45010	390 + 388	0	780
Ex. 3 rev.	20	17575 / 902 / 45010	397 + 404	0	802

returned by our computation. Thus, we look for ways to improve the usefulness of the computation by filtering out ‘uninteresting witnesses’ and keeping a more succinct yet informative set of witnesses.

To address this problem, we have defined and implemented a filtering mechanism. At every stage of the computation, given the set of witnesses that was already found, the mechanism supports the filtering of witnesses that (1) only include objects of classes instantiated in previously found witnesses (NNC), (2) only include types of associations appearing in previously found witnesses (NNA), and (3) only include combinations of classes and associations appearing in previously found witnesses (NNCA). For example, recalling Fig. 2, after om_1 is found, when using the first filter NNC, all additional object model diff witnesses consisting of only employees, tasks, and managers, would be filtered out from the results (thus, in this case, after om_1 is found, no more diff witnesses will be reported).

Table 3 shows the results of applying our filtering mechanisms to the diff witnesses computation of *cddiff*. We report on applying the filters to the examples shown in Sect. 2 and to the library example (the same examples considered in Sect. 5.3). Note that the cases where there are no diff witnesses are omitted from the table because they are irrelevant for the filtering issue.

Table 2. Results from computing diff witnesses for the library example, using different scopes. Each example is reported twice, computing the differences in both directions. Columns are the same as the ones in Table 1.

Name	Scope	Vars/p. vars/clauses	Alloy time (ms)	# Wit.	Plug-in time (ms)
V1 vs. V2	5	10735 / 429 / 28947	173 + 27	20	536
– rev. –	5	10735 / 429 / 28932	166 + 17	20	592
V1 vs. V2	10	42590 / 1544 / 120542	1370 + 68	20	2476
– rev. –	10	42590 / 1544 / 120512	1250 + 56	20	2338
V2 vs. V3	5	10947 / 429 / 29522	172 + 31	0	206
– rev. –	5	10947 / 429 / 29523	171 + 30	0	206
V2 vs. V3	10	43442 / 1544 / 123257	1344 + 109	20	2761
– rev. –	10	43442 / 1544 / 123258	1422 + 97	20	2432
V3 vs. V4	5	46347 / 1562 / 124413	1102 + 125	20	2135
– rev. –	5	46347 / 1562 / 124368	1093 + 219	20	2622
V3 vs. V4	10	120807 / 3997 / 331983	5583 + 812	20	9821
– rev. –	10	120807 / 3997 / 331903	5617 + 384	20	11103
V4 vs. V5	5	33380 / 1144 / 91631	678 + 173	20	1791
– rev. –	5	33380 / 1144 / 91617	674 + 66	20	1660
V4 vs. V5	10	93995 / 3199 / 263066	4016 + 780	20	8241
– rev. –	10	93995 / 3199 / 263042	4047 + 291	20	7051

In all cases we first ran *cddiff* without the filters and saw that it produces at least 20 diff witnesses. Then we ran it again, each time with a different filter. The table shows the effectiveness of the filters in significantly reducing the number of witnesses. Moreover, the remaining witnesses are guaranteed to be rather different from one another and thus interesting for the engineer. Note, however, that the effectiveness of these filters depends, to a certain extent, on the order in which Alloy finds the instances, which, unfortunately, is undefined. Thus, for example, we may end up with a different set of witnesses each time we run *cddiff* with the same two CDs as input. Also, a larger scope does not guarantee that we are left with more witnesses after filtering (see, e.g., in the last section of Table 3, increasing the scope from 5 to 10 for Lib. V4 vs. V5 rev. reduced the number of witnesses that passed the filters from 3/2/4 to 3/1/4).

The filters described above and are reported on in the experiments can be considered *incremental* or *online* filters, because they are applied to the results online, as they are found during the computation. Alternatively, we may suggest *static* filters, which take the complete set of all the computed diff witnesses (up to the given scope), apply a classification based on some criteria, and then output a representative witness from each equivalence class. For example, a possible criteria for classification may be the set of classes represented in the diff witness object model. Two diff witnesses would be considered equivalent if they contain object instances from exactly the same set of classes. This would ensure variability in the set of representatives that is included in the final output. Note that the witnesses provided by the incremental filter *NNC*, which we described above, can all be viewed as representatives of different equivalence classes. However,

Table 3. Results from applying filters to the examples shown in Sect. 2 and to the library example (the same examples considered in Sect. 5.3). Note that the cases where there are no diff witnesses are omitted from the table, because these are not relevant to the filters.

Name	Scope	# Wit.	# After filtering by NNC / NNA / NNCA
Ex. 1	5	20	2 / 3 / 3
Ex. 1 rev.	5	20	3 / 2 / 4
Ex. 1	10	20	2 / 2 / 3
Ex. 1 rev.	10	20	3 / 3 / 6
Ex. 2 rev.	5	20	2 / 1 / 3
Ex. 2 rev.	10	20	1 / 1 / 2
Lib. V1 vs. V2	5	20	5 / 4 / 5
Lib. V1 vs. V2 rev.	5	20	4 / 2 / 4
Lib. V1 vs. V2	10	20	2 / 2 / 3
Lib. V1 vs. V2 rev.	10	20	4 / 2 / 4
Lib. V2 vs. V3	10	20	4 / 3 / 6
Lib. V2 vs. V3 rev.	10	20	5 / 3 / 6
Lib. V3 vs. V4	5	20	3 / 2 / 4
Lib. V3 vs. V4 rev.	5	20	4 / 1 / 4
Lib. V3 vs. V4	10	20	4 / 3 / 4
Lib. V3 vs. V4 rev.	10	20	4 / 2 / 6
Lib. V4 vs. V5	5	20	3 / 2 / 4
Lib. V4 vs. V5 rev.	5	20	3 / 2 / 4
Lib. V4 vs. V5	10	20	3 / 2 / 4
Lib. V4 vs. V5 rev.	10	20	3 / 1 / 4

the alternative static filter variant is better, as its output may be more complete and include representatives of additional equivalence classes.

The use of the different filters in our plug-in is optional. Further evaluation of the effectiveness of these filters and the development of additional ones are left for future work.

6.2 Abstraction

Abstraction, a fundamental concept in model-driven engineering, has an important role in the context of CD comparisons. Specifically, two models may be equivalent at one level of abstraction but different in a less abstract level. Thus, the level of abstraction of interest should be defined by the engineer applying the comparison, who may be aware that the models at hand differ at a certain detailed level, but would be interested in comparing them at a higher level, where they are supposedly equivalent.

To this end, we have defined and implemented an *attribute abstraction*. With this abstraction in effect, *cddiff* ignores differences that are caused only by local changes to the attribute lists of the classes in the diagrams. That is, all class attributes of primitive or library types are abstracted away, so that two CDs whose sole difference is at the attributes level are considered equivalent. For

Table 4. Results from computing diff witnesses for the library example, with and without the attribute abstraction. Columns are the same as the ones in Table 1

Name	Scope	Vars/p. vars/clauses	Alloy time (ms)	# Wit.	Plug-in time (ms)
V2 vs. V3	5	10947/429/29522	168 + 30	0	202
– w./abs. –	5	10947/429/29522	165 + 39	0	207
V2 vs. V3	10	43442/1544/123257	1302 + 114	20	2652
– w./abs. –	10	43442/1544/123257	1371 + 108	20	2577
V3 vs. V4	5	46347/1562/124413	1123 + 97	20	2159
– w./abs. –	5	12952/486/34336	269 + 45	0	320
V3 vs. V4	10	120807/3997/331983	5547 + 731	20	9508
– w./abs. –	10	51302/1756/143631	1747 + 153	20	3161
V4 vs. V5	5	33380/1144/91631	673 + 173	20	1681
– w./abs. –	5	0/0/0	184 + 0	0	187
V4 vs. V5	10	93995/3199/263066	4091 + 802	20	8154
– w./abs. –	10	0/0/0	1540 + 0	0	1542

example, in Fig. 4, if an attribute ID is added to the class `Employee` (in only one of the CDs) or to the abstract class `Person`, the two CDs are still considered semantically equivalent under the attribute abstraction.

The attribute abstraction becomes useful when the engineer is aware of attribute-level differences resulting from local changes, but is interested in checking for more global semantic differences, if any. Another application of this abstraction relates to performance and scope. Given two large CDs, with many classes or many attributes per class, one can start by a comparison with the abstraction in effect. If a difference is found, indeed this proves that the CDs’ semantics are different. If a difference is not found, however, one has no choice but to make the comparison again with a higher scope or without the abstraction.

As a concrete example, we have compared the performance and completeness of *cddiff* with and without the attribute abstraction when running on CDs from the library example. Recall that in this example, each CD has 11 classes and the average number of attributes per class is 4. The results are shown in Table 4. On the one hand, the results show that the abstraction can reduce the size of the problem for Alloy and accelerate the computation of the diff witnesses. On the other hand, as expected, the analysis with abstraction is incomplete: in some cases it does not find all the diff witnesses that can be found without the abstraction. For example, the results for V3 vs. V4 with scope 5 show that 20 witnesses were found without abstraction, but none were found with abstraction. Interestingly, in the case of V4 vs. V5, the abstraction caused Alloy to construct an empty formula: the only differences between V4 and V5 are in some attributes and thus, without them, Alloy’s formula construction and minimization was able to directly reduce the differencing predicate to false. In contrast, in the case of V2 vs. V3, the size of the formula constructed by Alloy, with or without abstraction, was the same. This happened because the differences between V2 and V3 are all *not* in the attributes, and so the optimization we use, of removing same-name attributes from same-name classes (see the end of Sect. 4.3), has the same effect as the attribute abstraction.

Defining and implementing additional abstractions to be supported by *cddiff*, e.g., an abstraction based on the composition hierarchy between classes or the containment hierarchy of packages and classes, is left for future work.

7 Discussion and Future Directions

We discuss some limitations of our work and list related future work directions.

7.1 Bounded Analysis and the Small Scope Hypothesis

The use of Alloy, and consequently, encoding the problem of computing the diff witnesses as an instance of SAT, carries a significant price: all analysis is bounded to the user-specified scope. If a witness is found, we know the CDs' semantics are different; if no witness is found, we do not know whether the CDs have equal semantics or there still is a witness of a larger size. Recall that by size we mean the maximal number of objects in the object model. As a simple example, assuming a difference in multiplicities, between $*$ and $0..m$, a witness of size $< m$ does not exist. In this sense, the analysis is sound but incomplete. It is important to note, though, that for a given scope k , the analysis is sound and complete: if a witness of size $\leq k$ exists, it is found.

Nevertheless, our experience with CD, as well as an informal survey we have conducted by checking hundreds of CDs that appear in several textbooks and in different projects, e.g., the meta-model of the UML (available in [27]), showed us that while the number of classes and associations in large CDs can be high (we have seen examples of CDs with more than 100 classes), the multiplicities used on associations are typically $0..1$, 1 , $1..*$, and $*$. Multiplicities that use specific numbers greater than 1 (e.g., a polygon class that has $3..*$ sides, a panel that has $1..10$ buttons), are rather rare.

Thus, as the scope limitation is relevant mostly to the multiplicities, we adapt the *small scope hypothesis* of [13] to our problem domain, and suggest that in many cases, although the CDs involved may be large and include many classes and associations, witnesses for their differences could be rather small. Moreover, the optimization suggested at the end of Sect. 4.3 helps us in coping with reducing the size of the problem for Alloy.

Still, given large CDs, or diagrams with no object models of small size, a symbolic technique or an abstraction/refinement approach may be recommended and required in order to allow our analysis to scale (see also subsection 6.2). Alternatively, it may be possible to identify cases where one can formally prove that a certain scope is 'good enough', that is, it may be possible to find sufficient conditions on the two CDs that will guarantee that a bounded analysis in this case is as complete as an unbounded one. We leave these directions for future research work.

7.2 Integration With Operation-Based and Syntactic Differencing

Our approach to semantic differencing is state-based rather than operation-based (on the distinction between the two see [23]). That is, the input for *cddiff* consists

only of the two versions of the CD, and includes no information about the edit operations, if any, that have led from the first to the second version. Some works, however, concentrate on operation-based differencing, or take the two versions and aim to reconstruct a (shortest) series of edits (additions, deletions, updates) that leads from one version to the other (see the related work discussion in Sect. 8). Moreover, our approach to differencing is semantic, while most related comparison approaches are syntactic.

Thus, it may be useful to combine syntactic and operation-based differencing with state-based semantic differencing of class diagrams. For example, one may extend the applicability of semantic differencing in comparing diagrams whose elements have been renamed or moved in the course of evolution, by applying a syntactic matching (see, e.g., [9]) before computing a semantic differencing. This would result in a mapping plus a set of diff witnesses. As another example, one may find ways to use information extracted from syntactic differencing as a means to localize and thus improve the performance of semantic differencing computations.

We leave these ideas for future work.

8 Related Work

We discuss related work in the area of CD formal semantics and analyses and in the area of model and program comparisons.

8.1 CD Formal Semantics and Analysis

Class diagrams are part of the UML standard and are widely used for the modeling of the structure of object-oriented systems, in particular in model-driven design and development setups. As such, many researchers have discussed the semantics of class diagrams and considered related analysis questions.

A number of works consider various analysis problems related to class diagrams (see, e.g., [6,20,33]). These include the finite satisfiability problem, the consistency between UML models, the problem of class equivalence, the identification of implicit consequences etc. Some of these works use Description Logic (DL) as their underlying formalism, some use linear programming methods, while others include no implementation but present theoretical results about the decidability and complexity of the problems at hand. In contrast, we consider the specific problem of semantic comparison and the generation of diff witnesses. We provide a solution, in a bounded scope, using a reduction to an Alloy module and its analysis with a SAT solver.

Some previous works consider the use of Alloy for the analysis of class diagrams (see, e.g., [4,32]). These work focus on the formal definition of the transformation of a single CD to an Alloy module at the level of a meta-model and on the implementation of this transformation using a transformation language. Possible applications of the use of Alloy to analyze a given CD are not discussed in depth in these works. In contrast, as explained earlier, the input for

our transformation consists of two CDs, and it produces a single Alloy module whose all instances, if any, represent the required diff witnesses. Defining and implementing our transformation using QVT or other transformation language such as ATL [15] is possible, but is outside the focus of our work.

Finally, in another paper in this conference [19] we presented modal object diagrams (MOD), as an extension of classical object diagrams, and a related verification process, which verifies a CD against an MOD specification. MOD verification is implemented using a transformation to Alloy, whose input is a CD and an MOD. It is different than the one we use here for *cddiff*.

8.2 Model and Program Comparisons

Model and program differencing, in the context of software evolution, has attracted much research efforts in recent years (see [11,10,17,22,26,34]). In contrast to our work, almost all studies in this area, however, present syntactic differencing, at either the concrete or the abstract syntax level.

Alanen and Porres [1] describe the difference between two models as a sequence of elementary transformations, such as element creation and deletion and link insertion and removal; when applied to the first model, the sequence of transformations yields the second. Kuster et al. [17] investigate differencing and merging in the context of process models, focusing on identifying dependencies and conflicts between change operations. Engel et al. [10] present the use of a model merging language to reconcile model differences. Comparison is done by identifying new/old MOF IDs and checking related attributes and references recursively. Results include a set of additions and deletions, highlighted in a Diff/Merge browser. Mehra et al. [22] describe a visual differentiation tool where changes are presented using editing events such as add/remove shape/connector etc. Xing and Stroulia [34] present an algorithm for object-oriented design differencing whose output is a tree of structural changes, reporting differences in terms of additions, deletions, and moves of model elements, assisted by a set of similarity metrics. Ohst et al. [26] compare UML documents by traversing their abstract-syntax trees, detecting additions, deletions, and shifts of sub-trees.

As the above shows, some works go beyond the concrete textual or visual representation and have defined the comparison at the abstract-syntax level, detecting additions, removals, and shifts operations on model elements. However, to the best of our knowledge, no previous work considers model comparisons at the level of the semantic domain, as is done in our work.

Some works, e.g. [9,34], use similarity-based matching before actual differencing. As our work focuses on semantics, it assumes a matching is given. Matching algorithms may be used to suggest a matching before the application of semantic differencing. The result of such an integration would be a mapping plus a set of differentiating traces.

We are aware of only a few studies of semantic differencing between programs. Jackson and Ladd [14] presented a tool that summarizes the semantic diff between two procedures in terms of observable input-output behaviors. Apiwattanapong et al. [5] presented a behavioral differencing algorithm for object-

oriented programs based on an extended control-flow graph representation, and a tool called JDiff, which implements it in the context of Java. Finally, Person et al. [28] suggested to compute a behavioral characterization of a program change using a technique called differential symbolic execution. We focus on model comparison and not on program comparison. Also, while our work is somewhat similar to these works in terms of motivation, it is very different in terms of technology.

9 Conclusion

We presented *cddiff*, a semantic differencing operator for class diagrams. Unlike existing approaches to model's comparison, *cddiff* performs a semantic comparison and outputs a set of diff witnesses, each of which is an object model that is possible in the first CD and is not possible in the second. We have formally defined *cddiff*, described the technique to compute it, and demonstrated its application in comparing CDs within the Eclipse IDE. When applied to the version history of a given CD, *cddiff* provides a semantic insight into its evolution, which is not available in existing syntactic approaches.

We have implemented *cddiff* and applied it to several examples. We have extended the basic *cddiff* technique with a filtering mechanism that filters out 'uninteresting witnesses' and reports a more succinct yet informative set of witnesses to the engineer. We have also extended the basic *cddiff* technique with an attribute abstraction mechanism. This abstraction becomes useful when the engineer is aware of attribute-level differences resulting from local changes, but is interested in checking for more global semantic differences, if any. It is also useful in addressing the scope limitation and in improving *cddiff* performance.

We discussed a number of challenges and directions for future work in Sect. 7, including the development of heuristics to improve the performance of *cddiff* and allow it to scale. An interesting future work is to extend *cddiff* with support for abstraction beyond the attribute abstraction we have already defined and implemented. Another direction for future work is the integration of *cddiff* with existing approaches to matching and syntactic differencing, in particular as a means to improve its performance and the usefulness of its results to the engineers. The usefulness of *cddiff* to engineers, in particular in comparison with existing syntactic approaches, should be empirically evaluated.

Finally, in a recent paper [18] we have described our more general vision on semantic model differencing. Thus, *cddiff* is part of a larger project [31], aiming to apply the idea of semantic differencing and the computation of diff witnesses to other modeling languages, including, e.g., activity diagrams, statecharts, and feature diagrams. We hope to report on our work in these directions in future papers.

Acknowledgments. We are grateful to Martin Schindler for defining the MontiCore language support for CDs. We thank Smadar Szekely and Guy Weiss for their expert advice on Eclipse plug-in development. We thank Eric Bodden, David Lo, and the anonymous reviewers for comments on a draft of this paper.

References

1. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
2. Alloy Analyzer website, <http://alloy.mit.edu/> (accessed April 2011)
3. Altmanninger, K.: Models in Conflict – Towards a Semantically Enhanced Version Control System for Models. In: Giese, H. (ed.) MODELS 2008. LNCS, vol. 5002, pp. 293–304. Springer, Heidelberg (2008)
4. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. *Software and Systems Modeling* 9(1), 69–86 (2010)
5. Apiwattanapong, T., Orso, A., Harrold, M.J.: JDiff: A differencing technique and tool for object-oriented programs. *Autom. Softw. Eng.* 14(1), 3–36 (2007)
6. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. *Artif. Intell.* 168(1-2), 70–118 (2005)
7. Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Definition of the System Model. In: Lano, K. (ed.) UML 2 Semantics and Applications, pp. 61–93. Wiley, Chichester (2009)
8. Cengarle, M.V., Grönniger, H., Rumpe, B.: System Model Semantics of Class Diagrams. *Informatik-Bericht 2008-05*, Technische Universität Braunschweig (2008)
9. EMF Compare, <http://www.eclipse.org/modeling/emft/?project=compare> (accessed April 2011)
10. Engel, K.-D., Paige, R.F., Kolovos, D.S.: Using a Model Merging Language for Reconciling Model Versions. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 143–157. Springer, Heidelberg (2006)
11. Evans, A., France, R.B., Peng, S.-L.: The UML as a Formal Modeling Notation. In: Bézivin, J., Muller, P.-A. (eds.) UML 1998. LNCS, vol. 1618, pp. 336–348. Springer, Heidelberg (1999)
12. Harel, D., Rumpe, B.: Meaningful Modeling: What’s the Semantics of “Semantics”? *IEEE Computer* 37(10), 64–72 (2004)
13. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
14. Jackson, D., Ladd, D.A.: Semantic Diff: A Tool for Summarizing the Effects of Modifications. In: ICSM, pp. 243–252. IEEE Computer Society, Los Alamitos (1994)
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. *Sci. Comput. Program.* 72(1-2), 31–39 (2008)
16. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer (STTT)* 12(5), 353–372 (2010)
17. Küster, J.M., Gerth, C., Engels, G.: Dependent and Conflicting Change Operations of Process Models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 158–173. Springer, Heidelberg (2009)
18. Maoz, S., Ringert, J.O., Rumpe, B.: A manifesto for semantic model differencing. In: Dingel, J. (ed.) MODELS 2010 Workshops. LNCS, vol. 6627, pp. 194–203. Springer, Heidelberg (2011)
19. Maoz, S., Ringert, J.O., Rumpe, B.: Modal object diagrams. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 282–306. Springer, Heidelberg (2011)
20. Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 17–31. Springer, Heidelberg (2007)

21. Massoni, T., Gheyi, R., Borba, P.: A UML Class Diagram Analyzer. In: CSDUML, pp. 143–153 (2004)
22. Mehra, A., Grundy, J., Hosking, J.: A generic approach to supporting diagram differencing and merging for collaborative design. In: ASE, pp. 204–213. ACM, New York (2005)
23. Mens, T.: A state-of-the-art survey on software merging. *IEEE Trans. Software Eng.* 28(5), 449–462 (2002)
24. MontiCore project, <http://www.monticore.org/>
25. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S.M., Zave, P.: Matching and merging of statecharts specifications. In: ICSE, pp. 54–64. IEEE Computer Society, Los Alamitos (2007)
26. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. In: Proc. ESEC / SIGSOFT FSE, pp. 227–236. ACM, New York (2003)
27. OMG. Unified Modeling Language (UML) Superstructure v2.1.2, <http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF> (accessed April 2011)
28. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential Symbolic Execution. In: SIGSOFT FSE, pp. 226–237. ACM, New York (2008)
29. Rumpe, B.: *Modellierung mit UML*. Springer, Heidelberg (2004)
30. SAT4J project, <http://www.sat4j.org/> (accessed April 2011)
31. Semantic diff project, <http://www.se-rwth.de/materials/semdiff>
32. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and Back Again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010)
33. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
34. Xing, Z., Stroulia, E.: Differencing logical UML models. *Autom. Softw. Eng.* 14(2), 215–259 (2007)

A Refactoring Constraint Language and Its Application to Eiffel

Friedrich Steimann, Christian Kollee, and Jens von Pilgrim

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen

steimann@acm.org, {Christian.Kollee,Jens.vonPilgrim}@feu.de

Abstract. We generalize previous work on constraint-based refactoring and develop it into the definition of a constraint language allowing the specification of refactorings in a completely declarative way. We present a compiler that transforms specifications in our language to plug-ins for an IDE that, together with an accompanying framework providing the necessary infrastructure, implement the specified refactoring tools. We evaluate our approach by presenting specifications of three different refactorings for the Eiffel programming language, and by applying the resulting refactoring tools to several sample programs. Outcome suggests that our approach is indeed viable.

1 Introduction

Constraint-based refactoring has proven to be a powerful approach to mastering the complexity of various type and accessibility related refactorings [4, 9, 14, 15, 26, 29, 30]. The declarative nature of constraints and of the rules that generate them allow one to stay close to the language specification, and away from the combinatorial explosion of possible expressions whose correct handling makes imperative formulations of the preconditions and the mechanics of a refactoring such a daunting task. However, we observe that the scope of constraint-based refactoring has so far unnecessarily been constrained to treating a single aspect of a programming language — such as typing or accessibility — in isolation, where the same approach, with some extensions, could be used to solve a much broader class of refactoring problems, including the joint handling of different language aspects in one refactoring. Preparing the ground for such a broadening of scope is the aim of our work presented here.

More concretely, we count six main contributions in this paper (and ask the reader to accept the following enumeration as an outline also):

1. We generalize previous work on constraint-based refactoring, extending it to the control of properties beyond types and accessibilities, allowing their joint treatment within a single refactoring (Section 3).
2. We identify a number of challenges of constraint-based refactoring uniformly posed by different programming languages, providing solutions that remain entirely in the realm of constraint satisfaction and thus a single solution framework (Section 4).

3. We specify a powerful, yet concise constraint language, called REFACOLA, allowing the completely declarative specification of various refactorings (Section 5.1).
4. We present an implementation of that language allowing the generation of refactoring tools for various platforms directly from their specifications (Section 5.2).
5. We present the REFACOLA specifications of three declaration-related refactorings for the Eiffel programming language (Section 6), showing that the constraint-based approach is capable of dealing with problems rather different from those found in Java (to which constraint-based refactoring has so far exclusively been applied).
6. We evaluate the computational cost of constraint-based refactorings resulting from REFACOLA specifications by applying them to a body of Eiffel programs (Section 7), showing the feasibility of our approach.

We have chosen Eiffel for our evaluation, and thus decided to break with the Java monotony that currently characterizes the field, because (a) continued concentration on a single language tends to draw attention to accidental problems caused by that language's design, rather than the essential problems of refactoring, (b) Eiffel is sufficiently different from Java and its kin to support our claim of generalizing constraint-based refactoring, and (c) because the Eiffel standard [6] makes heavy use of so-called validity rules, a form of language specification that has a rather direct mapping to the constraint rules of constraint-based refactoring, thereby increasing one's belief in that refactoring tools correctly incorporate the language specification.

2 Motivation and Related Work

Although the discipline of refactoring — as initiated by the works of Griswold [11] and Opdyke [21] — is already almost two decades old, currently available refactoring tools are still plagued with bugs (see, e.g., [3, 22, 24–26]). While some would maintain that refactorings are used mainly in agile settings characterized by excessive regression testing anyway [8], we object that refactoring tools are *metaprograms* that play in the same league as editors, compilers, and version control, for the correctness of which one too would not want to rely on testing one's own *object programs*.

Roughly, the symptoms of today's refactoring tools' failures can be divided into four categories:

1. A refactoring that is possible is nevertheless refused.
2. The refactored program contains errors (compile time or runtime) the original program did not exhibit.
3. The refactored program behaves differently from the original program.
4. The refactored program does not exhibit the change of structure reflecting the refactoring intent.

Except for the last, all problems can be blamed on a partial ignorance, on behalf of the refactoring tool, of the target language's syntax and semantics, and thus should be resolvable, once and for all, by incorporating the relevant parts of the language specification into the tool's refactoring procedure.

One approach to making a refactoring language-specification aware is to specify it in terms of rewrite rules to be applied to a graph representation of programs [18],

where the rewrite rules are designed to observe the syntactic and semantic rules of the programming language. However, it turns out that the necessary precondition checking, and also the correctness-preserving transformations themselves, are nontrivial to formulate using graph grammars, which is why these approaches resort to considerable imperative components (“programmed graph grammars”) that suffer the same problems as the standard procedural implementations of refactorings. As far as we know, no widely disseminated refactoring tools as yet utilize this approach.

ASTGen [3] uses comprehensive syntactic and (static) semantic knowledge of the programming language whose programs are to be refactored, for generating programs that provoke errors in refactoring tools. While this proved to allow for very effective testing of existing refactoring implementations, it is not obvious how the language-awareness of ASTGen could be incorporated directly into a refactoring tool.

In a series of works, Tip et al. [9, 15, 30, 29] have used constraints to include Java’s typing rules into the precondition checking and mechanics of refactoring tools, thereby avoiding errors of above category 2. Ref. 26 has picked up the approach and transferred it to the rules of access control in Java. However, as has been pointed out in [26, 29], obeying the typing and accessibility rules alone is insufficient to prevent errors of category 3: in presence of overloading, hiding, and other name resolution issues, constraints modelling the name lookup in Java may be needed which, given its imperative nature, are hard to express. That name binding rules even for a language such as Java can be modelled as constraints (although admittedly at some expense) has been shown in [26, 27].

In a complementary series of works, Schäfer et al. [22–25] have elaborated how to preserve, among other dependencies, the binding of references (names) to declared entities. For this, they have established a notion of *locked names* which can be conceived of as pointers to declared entities persisting all refactorings. Once a refactoring has been performed, the locked names are converted back to Java names, inserting qualifiers where necessary to force the original binding. While this procedure does not always succeed (there may be cases for which no suitable qualification exists, or an entity involved in a qualification may be inaccessible), it has proven well-suited for a language such as Java with its intricate problems of hiding, shadowing, and obscuring [12], which are hard to deal with otherwise. However, it is far less useful for a language like Eiffel, in which scopes cannot be nested and the only possibility to resolve name conflicts is by renaming.

While all of the above works focus on implementing refactoring tools, JunGL [32] is a refactoring scripting language providing the necessary infrastructure for such implementations. Like [18], it builds on a graph representation of programs, but adds powerful querying facilities based on a combination of functional and logic programming (Datalog). However, the mechanics of the refactorings must still be specified imperatively, limiting declarativeness of the approach to precondition checking.

Constraint languages (which are inherently declarative) have so far mostly been devised for other problems than refactoring. For instance, CCEL was designed as a constraint expression language for C++ allowing the definition of design, implementation, and stylistic rules that programmers should obey [20]. A CCEL constraint consists of a set of universally quantified variables and an assertion that uses the variables. Constraints are applied to programs by binding the variables to matching components of the program, and by evaluating the constraints for each found binding,

with a violated constraint indicating a violated rule. A similar approach is taken by more recent work on pluggable type systems [1], which also uses a constraint language (called JavaCOP). While bearing many similarities to the constraint rules that constraint-based refactoring rests on, neither CCEL nor JavaCOP have notions of allowing a program to change in such a way that its components meet the required constraints — this however is a necessary feature of every refactoring language.

3 A Generalized Framework of Constraint-Based Refactoring

A common tenet of all constraint-based refactorings is that for the purpose of a specific refactoring, a program is sufficiently represented by a set of constraint variables and a set of constraints that relate their values, where both are derived from the program to be refactored by application of so-called constraint rules [4, 9, 26, 29, 30]. Depending on the concrete refactoring, the constraint variables represent properties of program elements such as types [29], type parameters [4, 9, 15], or declared accessibility [26], but generally, there is no reason to restrict constraint variables to one kind of program properties per refactoring, just as there is no general limit as to which properties can at all be represented by constraint variables. This observation is the starting point of our conception of a refactoring constraint language.

3.1 Program Elements, Kinds, and Properties

For our purposes, it is sufficient to assume that a program consists of

- a set D of *declared entities* (elsewhere also referred to as *definitions*),
- a set R of *references* to declared entities (also called *uses* of definitions), and
- relationships on subsets of $D \cup R$.

We refer to declared entities and references collectively as *program elements*.

Program elements come in different *kinds*. Kinds that will be found in most object-oriented programming languages are *Class*, *Field*, *Method*, and *Local* for classes, fields, methods, and locals (formal parameters and local variables), respectively, and *Reference* for references to (uses of) declared entities; but generally, different languages may have different kinds of program elements.

Depending on the programming language, different kinds of program elements have different *properties* such as identifiers, types, declaring classes, etc. Deviating from prior work on constraint-based refactoring, which considered only a single property each (and therefore could afford to use brackets to denote types [29] or accessibilities [26]), we do not limit the kind of properties considered by any single refactoring (and therefore use Greek letters to denote properties, writing $e.\pi$ for the property π of an element e). As will be seen, it is useful to organize the kinds of program elements in a subsumption hierarchy and to let subkinds inherit the properties of their superkinds.

It is important to note that a refactoring will usually alter properties of program elements and not the program elements themselves — the elements' identity is always preserved under refactoring (meaning that an element cannot become another one; in fact, it is legitimate to think of program elements as objects in the object-oriented

sense, as well as of kinds as classes, and of properties as fields). It is the properties, and not the program elements, that are mapped to the constraint variables of a constraint-based refactoring.

3.2 Domains

Each *property* is associated with a *domain* from which its possible *values* are drawn. Although the domains of properties are essentially the domains of constraint variables most of which will be represented by the set of integers (see below), at the refactoring constraint language level it is useful to distinguish different domains of properties. For instance, the domain of a property *identifier* (ι), *Identifier*, is the set of all valid identifiers of a programming language, and the domain of the property *accessibility* (α) in Java is the set $\{\textit{private}, \textit{package}, \textit{protected}, \textit{public}\}$. Domains are essentially types, and constraints are typed; a constraint such as $r.\iota = d.\alpha$, although solvable by the solver (if the integer representations of the corresponding domains overlap), does not make sense.

Program-Dependent Domains. Some domains' members are drawn from the elements of a program. For instance, the value of a *location* property that associates a method with its declaring class is a class, which is also program element (of kind *Class*). Since properties are constraint variables, and since standard constraint solvers know nothing of program elements as defined here, we cannot use kinds as the domains of properties. Instead, we allow domains of properties to be based on kinds, by introducing an injective mapping $[\cdot]$ from kinds to domains, writing $[K]$ for the domain corresponding to kind K and (by extending the mapping from kinds to their elements) $[e]$ for the value corresponding to program element $e \in K$.

Ordered Program-Dependent Domains. Program-dependent domains such as $[Class]$ may be (partially) ordered: in this case, the ordering relation has to be extracted from the program from which the domain is drawn. In the example of $[Class]$, one ordering relation is the subclass relation; another would be the nesting of classes (if allowed by the language).

3.3 Constraints, Constraint Rules, and Constraint-Based Refactoring

The constraint variables and constraints that represent a refactoring problem are generated from the program to be refactored by application of so-called *constraint rules*. Each constraint rule is of the general form

$$\frac{\textit{program queries}}{\textit{constraints}}$$

where the rule precedent is a logical expression (see below) selecting the program elements for which the rule is to take effect, and the rule consequent lists the constraints to be generated. The program queries contain variables (which are not constraint variables!) that are bound to program elements during application of the rule; the properties of these program elements make the constraint variables of the constraints generated by the application. Each rule is implicitly universally quantified

over its variables representing program elements, and application of the rule to a program will find all combinations of program elements matching the queries expressed in the rule precedent. For instance, if the query $binds(r, d)$ of the rule

$$\frac{binds(r, d)}{r.\iota = d.\iota \quad r.\tau = d.\tau}$$

finds the pairs (r_1, d_1) and (r_2, d_2) , the constraints $r_1.\iota = d_1.\iota$, $r_1.\tau = d_1.\tau$, $r_2.\iota = d_2.\iota$, and $r_2.\tau = d_2.\tau$ will be generated and added to the constraint set (note how the rule involves both identifier, ι , and type, τ , properties). Adjacent expressions above and below the bar are implicitly conjoined; explicit logical junctions are also possible (examples will be given in the following sections).

A constraint system generated from a program by application of the constraint rules is always solved with the variable values reflecting the program as is (the *initial values*); it may however be invalidated by assigning one or more variables new values to reflect the goal of the refactoring. Solving the invalidated constraint system then amounts to computing the additional changes mandated by the refactoring. Note how this blurs the usual distinction between precondition checking and the mechanics of a refactoring: an unsatisfiable constraint system reflects a precondition violation.

3.4 Program Queries and Writing Back Solutions

The program queries constituting the precedents of the constraint rules are basically Datalog-like [2, 32] expressions whose predicates are matched against a (thought) fact base representing the program to be refactored. Every predicate has a name and a number of variables as arguments; variables occurring in more than one predicate refer to the same program element. Evaluating these queries produces all tuples of program elements satisfying the rule precedent. How the queries are evaluated is outside the scope of this paper; in practice, queries will be transformed to searches of the AST of the queried program, but generating an intermediate representation of the program and storing it in a database to speed up querying is also feasible.

Once the constraints have been generated and solved, the variable assignments that constitute the solution must be translated to necessary changes of the program to be refactored, a process that we refer to as *writing back the solution*. Depending on the property a constraint variable represents, writing back amounts to changing a component of a declaration (such as identifier, type, accessibility, or other modifiers) or a change of location (declaring class) of a program element, or it may lead to the introduction or deletion of code (a novel aspect that will be detailed in Section 4.4).

3.5 Specifying a Refactoring

If the constraint rules identified for a given programming language are sufficient to guarantee meaning preservation for all possible changes of the constrained program properties, every solution of the constraint system generated from a given program corresponds to a refactoring of that program. Specification of a concrete refactoring

such as `RENAME` or `GENERALIZE DECLARED TYPE` therefore amounts to narrowing the solution space to the solutions reflecting the changes associated with that refactoring, that is, the *refactoring intent*.

Specification of a concrete constraint-based refactoring usually involves specifying

- the kinds of program elements the refactoring can be applied to (e.g., all declared entities in the case of `RENAME`, or typed entities in the case of `GENERALIZE DECLARED TYPE`),
- the properties that are to be changed by the refactoring (e.g., identifiers in the case of `RENAME`, and declared types in the case of `GENERALIZE DECLARED TYPE`), as well as other properties whose adjustment may make a concrete refactoring possible (such as the identifiers of other program elements, e.g. references), and
- to which (sets of) values the changeable properties may be changed by the refactoring (e.g., a supertype in the case of `GENERALIZE DECLARED TYPE`).

All this information is to be supplied by the *author of the refactoring*. A concrete application of the refactoring further involves specification of

- the concrete program element(s) to which the refactoring is to be applied and
- the target values of the specified element(s)'(s) properties.

This information is to be supplied by the *user of the refactoring*.

4 Challenges of Constraint-Based Refactoring

While the previous section identified the basic constituents of a constraint-based refactoring framework, this section deals with specific problems that we encountered during our work on implementing constraint-based refactoring tools using this framework. The nature of these problems and their repeated occurrence in various refactorings that we have worked on gave rise to the definition of `REFACOLA` as will be presented in Section 5. One such problem, the *handling of foresight* (necessary for `MOVE` refactorings that change the AST of a program) requires techniques whose presentation exceeds the spatial limits of this paper; it is presented in a companion paper [27].

4.1 Indirection

Specifying the semantics of programming languages as constraints occasionally involves indirection. For instance, in Java for a member declared *package local* to be accessible by a reference r on receiver q , the type t of q must be defined in the same package the reference is located in, as would be expressed by the constraint $t.\pi = r.\pi$ (in which π denotes the *package* property). However, if the type t of the receiver q is variable (because a refactoring such as `GENERALIZE DECLARED TYPE` [29] is allowed to change it), this constraint does not model the problem adequately, since t , which is bound to a program element during application of a constraint rule (cf. Section 3.3), during constraint solution always denotes the same type, no matter whether the type of q , $q.\tau$, is changed by a refactoring. What we would really need to express is

$$q.\tau.\pi = r.\pi \tag{1}$$

in which $q.\tau$ denotes the property representing the type of q , and $q.\tau.\pi$ denotes the property representing the package of the value of $q.\tau$. Note that which property $q.\tau.\pi$ denotes depends on the value of $q.\tau$, which is itself a property (and thus variable). Thus, the package property of the receiver type is indirectly accessed.

As detailed in Section 3.2, the members of the domain of a property cannot themselves have properties, so that indirections of the above kind cannot be expressed directly. However, with the aid of program-dependent domains (here: $[Type]$ as the domain of $q.\tau$), the indirection of (1) can be expressed as a quantified constraint [27]

$$\exists t \in Type: q.\tau = [t] \wedge t.\pi = r.\pi \tag{2}$$

which is satisfied only if there is a program element t that is located in the same package as r and whose corresponding program-dependent domain value $[t]$ is the value of $q.\tau$. It turns out that (2) directly maps to the *element* constraint [17, 31] offered by many constraint solvers: if i and x are integer variables and A is an array of integer variables, *element*(i, A, x) is satisfied if and only if the constraint $A[i] = x$ is satisfied. For instance, using *element* the indirection constraint (1) can be expressed as

$$element(q.\tau, \Pi, r.\pi)$$

in which Π is an array of package properties such that $\Pi[i]$ is $t_i.\pi$, the package property of type t_i whose corresponding value $[t_i]$ is encoded (through the mapping of the domain $[Type]$ to the set of integers; cf. Section 3.2) by integer i , and in which $q.\tau$ (whose domain is $[Type]$) serves as the index into Π .

4.2 Reducing the Solution Space: Generating the Necessary Constraints Only

The usual approach to constraint-based refactoring is to apply all constraint rules to all elements of a program to be refactored, thereby generating huge numbers of constraints, including ones for program elements that are not involved in an intended refactoring. Depending on how the generated constraint system is solved, this procedure may not only waste resources in constraint generation, it can also produce myriads of solutions the user of the refactoring did not request, or has no interest in.

As stated in Section 3.5, input to the concrete application of a refactoring are the variable assignments that reflect the refactoring intent, and the set of other variables (properties) whose value may be changed by the refactoring. Of the latter, only the variables participating in constraints that may be invalidated by the former are of interest; all other variable assignments represent solutions that are independent from the refactoring intent, and should therefore not be generated. An algorithm that determines which constraints and constraint variables are to be generated to constrain precisely these properties is shown in Figure 1.

The algorithm is straightforward. It starts with the set of properties whose value must change, determines all constraints derivable from the program that directly constrain their new values, and replaces the occurrences of the properties in the constraints with constants reflecting the new values. For all other properties constrained by these constraints, it checks for each one whether it may be changed by the

Algorithm *GenerateConstraints***Input:**

M , the set of properties whose values must change
 N , the set of properties whose values may change
 v_0 , a function mapping properties to their initial values
 v_f , a function mapping the members of M to their new values
 R , a set of constraint rules

Output:

C , a set of constraints

Steps:

```

1. let  $P = M$ ,  $Q = \emptyset$ ,  $C = \emptyset$ 
2. repeat
3.   move one property  $p$  from  $P$  to  $Q$ 
4.   for each constraint rule  $r$  in  $R$ 
5.     if  $r$  can constrain  $p$ 
6.       for each constraint  $c$  generated by  $r$ , constraining  $p$ 
7.         for each property  $p'$  occurring in  $c$ 
8.           if  $p' \in M$ 
9.             replace every occurrence of  $p'$  in  $c$  with  $v_f(p')$ 
10.          else if  $p' \in N$ 
11.            replace every occurrence of  $p'$  in  $c$  with  $v_0(p')$ 
12.          else if  $p' \notin Q$ 
13.            add  $p'$  to  $P$ 
14.          add  $c$  to  $C$ 
15. until  $P = \emptyset$ 

```

Fig. 1. Algorithm computing the set of constraints to be generated for a specific refactoring

refactoring: if not, it replaces its occurrences in the constraints with the property's initial value (again a constant); otherwise, it applies the procedure recursively to the property, i.e., it determines which other constraints constrain the property (thereby indirectly constraining the properties to be changed) and so forth. All constraints determined by this algorithm constrain, either directly or indirectly (through properties that may be changed), the properties that must be changed, and are therefore necessary; all others can be dispensed with. The constraint variables generated by the algorithm are those constrained by its generated constraints.

4.3 Determining the Best Solution: Soft Constraints

Even with the algorithm of Figure 1 in place, a constraint system generated from a refactoring problem has usually more than one solution. The first solution produced by a general purpose constraint solver depends on internals of the solver, and may not be the one that a custom algorithm tailored for the specific problem computes (that is, the "best" solution).

To find a best solution, contemporary solvers allow the definition of an objective variable (also called a soft constraint) whose value they will then maximize or minimize. This can be exploited for refactoring problems, namely by defining an objective variable as accumulating *penalties* for violating certain aptness criteria to be respected by a good solution, such as changing as few properties as possible or preferring insertion of a cast at the call site of an overloaded method over renaming it (see below). In general, penalties can be expressed as equality constraints of the form

$$c_i = \text{if } \textit{constraint}(\pi_i, \dots) \text{ then } p_i \text{ else } 0$$

in which c_i is a constraint variable representing the cost of property π_i having a certain value (which one being specified by $constraint(\pi_i, \dots)$, which may involve other properties including the original value of π_i) and in which p_i is an integer constant representing the penalty. The solver must then be instructed to minimize an objective variable c whose value is constrained to the sum of all c_i .

Soft constraints come at a high price, however: in the worst case, finding the minimum requires exploration of the complete solution space, which is generally exponential in the number of constraint variables. Therefore, specifying penalties must be carefully traded against local search strategies that try to solve a constraint system starting from a solution that was invalidated by changing one or more variable values.

4.4 Defaults and Introductions

Sometimes the solution of a refactoring problem requires the introduction of a new program element. For instance, in the Java program

```
class Server {
    boolean m(String s) { return s.length() > 0; }
    void m(Comparable s) {...}
}
class Client {{ new a.Server().m("abc"); }}
```

the parameter type of `Server.m(String)`, `String`, can be generalized to `CharSequence` without affecting the method's well-typedness, allowing instances of other implementations of `CharSequence` to be passed. However, this will make the client's invocation of `m` ambiguous unless a cast to `CharSequence` is inserted before the actual parameter "abc". Since a constraint solver cannot introduce new program elements, we have to assume that such a cast, c , is already present in the program, only that it is not visible in the program text, because the cast's target type property, $c.\tau$, has a default value, namely the type of the expression to be cast (here: `String`). Should the solver assign such a "hidden" property a different value (different than the default), this value will materialize in the program text once the solution of the constraint system is written back (Section 3.4). The sparing use of non-default values (i.e., use only if the refactoring is impossible otherwise, or requires even less apt changes) can be enforced by corresponding soft constraints (penalties; see above), in the above example by penalizing $c.\tau \neq r.\tau$, in which r represents the reference to be cast by the (hidden) cast c .¹

4.5 Miscellaneous

To save space, we omit a number of other challenges and their solutions here. This includes the handling of external program elements such as those contained in libraries (which may constrain a refactoring, but may not themselves be subject to change),

¹ Note that inserting a cast is not the only possibility to fix the presented refactoring problem: other constraints of the program permitting, accessibility of the method competing in the binding, `Server.m(Comparable)`, could be lowered to `private`, or one of the two methods could be renamed. It is one of the strengths of our approach that if constraints for all three properties are generated, the refactoring will consider all possible solutions, and will choose the one imposing the least cost (in terms of the penalties defined).

the reduction of large domains such as *Identifier* or *[Class]* to smaller ones that do not contain values leading to equivalent solutions (used for RENAME FEATURE in Section 6.1), and the modelling of orderings that are themselves subject to change by a refactoring (such as class and type hierarchies for refactorings like EXTRACT INTERFACE [29] or REPLACE INHERITANCE WITH DELEGATION [14], or nesting of program elements for MOVE [8] refactorings). However, we can assure the reader that all these challenges can be solved by suitable encodings of the involved domains.

5 The Refactoring Constraint Language REFACOLA

Based on the generalized framework presented in Section 3 and on our solutions to the challenges identified in Section 4 we have devised REFACOLA, a *refactoring constraint language* allowing us to express the constraint rules specific to a programming language, as well as to specify refactorings relying on these rules, in a declarative way. Examples of specifications in REFACOLA can be found in the figures below; because of a lack of space, we do not present its syntax rules here. More information on REFACOLA can be found under <http://www.feu.de/ps/prjs/refacola>.

5.1 The REFACOLA Language and Framework

As can be seen in Figure 2, the REFACOLA language has three kinds of modules: languages, rulesets, and refactorings. A *language module* consists of the definitions of kinds (of program elements; cf. Section 3.1), their properties, domains, and the signatures of program queries. The kinds are arranged in a subkind hierarchy (using <:) and the properties associated with each kind (if any; appended to it in curly braces) are inherited by its subkinds. Properties are associated with domains (separated by a colon), which are either predefined (such as *Identifier* or *Boolean*) or enumerated (such as *Accessibility* in Java or C#) or program-dependent (such as *[Class]*; cf. Section 3.2). Properties with the same domain are compatible: for instance, class and type properties can be used interchangeably if the domain of both is *[Class]*.² The (partial) ordering of program-dependent domains is specified by referring to a program query extracting the ordering from the program to be refactored. For instance, in Eiffel the domain *[Class]* has two partial orders, one being defined by a query *inherits* and the other by a query *inherits-non-conforming* [6 §8.6.9]. For the first ordering defined, *<=* and *<* can be used as relation symbols (see, e.g., Figure 5); for all that follow, the query predicate defining the order has to be used. Queries are defined as signatures only; they provide the interface to the querying component of the REFACOLA framework (see below). *ENTITY* and *REFERENCE* are predefined kinds corresponding to *D* and *R*, respectively (cf Section 3.1).

A *ruleset module* imports a language module (using the *for* keyword) and consists of a set of rules, each carrying an identifier. The specification of a rule is comprised of the declaration of the variables serving as placeholders for program elements (introduced by *for all*), an *if*-part for the rule precedent, and a *then*-part for the rule consequent. The precedents of rules are Datalog-like [2] expressions whose atoms are

² Note that, for the sake of simplicity, we ignore parameterized types throughout this paper; that they can be treated with constraints has been shown elsewhere, e.g. in [4, 9, 15].

```

Language Eiffel
kinds
  Deferrable <: ENTITY {deferred}
  Class <: Deferrable
  Assignable <: REFERENCE {type}
  Feature <: Deferrable
properties
  deferred : Boolean
  type : Type
domains Type = [Class] ordered by inherits
queries
  has(Class, Feature) "Class defines or redefines or inherits Feature"
  create(Assignable) "object of Assignable's type is created and assigned to it"
  inherits(Class, Class) "1st Class inherits from 2nd"

ruleset Deferring for Eiffel
rules
  deferred-class
    for all c : Class f : Feature
    if has(c, f)
    then f.deferred = true implies c.deferred = true
  instantiation
    for all a : Assignable
    if create(a)
    then a.type.deferred = false

refactoring ChangeDeferredState for Eiffel uses Deferring
forced changes deferred of declaredEntity
allowed changes deferred of Class

```

Fig. 2. Sample language, ruleset, and refactoring modules for the deferred property (corresponding to abstract in Java et al.) of classes and features (members) in Eiffel. The rule named instantiation constrains the type of an (variable or attribute) to one that is instantiable (i.e., not tagged as deferred), if the assignable is used in an instance creation expression ([6 §8.20.6]). Note that this rule involves indirection (Section 4.1); this is necessary if the type of an assignable is subject to change by a refactoring (such as CHANGE DECLARED TYPE; see Section 6.3). The ordering of the program-dependent domain Type is defined by the inherits query; the ordering is not used in this example.

program queries of the imported language module; the consequents are sets of constraints that are to be generated for the program elements to which the rules apply.

The constraints supported by REFACOLA are (currently) those supported by our used constraint solver (Cream [28]) and include program-dependent orderings (using the Relation constraint [28] with the relations extracted as described above) and indirection (Section 4.1).

A *refactoring module* imports a language and one or more ruleset modules of the same language, and specifies which kind of program element the refactoring is to be applied to, which properties it is to change (the “forced changes”), and also which other properties of which other elements (if any) may be changed in the course of the refactoring (the “allowed changes”; cf. Section 3.5).

Based on a refactoring module and the language and ruleset modules it depends on, the REFACOLA compiler generates code that plugs into the REFACOLA framework that is itself embedded in an IDE such as Eclipse, MonoDevelop, or EiffelStudio (see Figure 3; currently, only Eclipse is supported, with adapters to the ASTs of MonoDevelop and EiffelStudio). The framework provides the implementations of the program queries, the constraint generator (which includes an implementation of the *Generate-Constraints* algorithm of Figure 1), and the routines necessary for writing back the solutions into refactored programs. Currently, small adapters are still hand-crafted to map the language definitions specified in REFACOLA to the types of the AST. Also, a

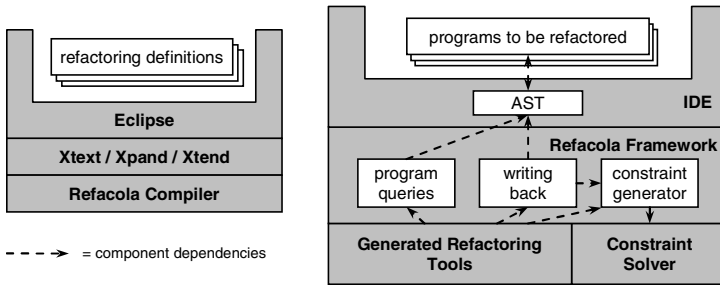


Fig. 3. REFACOLA compiler and framework and their embeddings in IDEs

full-fledged implementation of the framework on the target platform would adapt the refactorings to the IDE’s user interface. As of today, however, we have only implemented the batch application required to produce the results presented in Section 7.

5.2 Implementation of the Compiler

Our REFACOLA compiler is implemented using the Xtext [34] language development framework, which is itself Eclipse-based [5] (cf. Figure 3). The necessary type checks (e.g., that properties are only qualified by program elements for which they have been defined, and that constraints and queries are supplied with properties of the required domains) are performed using xtext-typesystem [35]. The code fragments (“plug-ins”) that — together with the REFACOLA framework and a constraint solver providing the necessary infrastructure — implement the refactorings specified in REFACOLA are generated using Xpand [33] and are based on templates specific to the programming language and target IDE.

6 Application

To demonstrate the expressiveness of REFACOLA, we first present specifications of the RENAME, CHANGE ACCESSIBILITY, and CHANGE DECLARED TYPE refactorings for the Eiffel programming language, and then explain how they are combined into a single refactoring whose applicability exceeds that of its components. The applicability of the refactorings is evaluated in Section 7.

6.1 The RENAME Refactorings

Eiffel has no explicit namespaces, so that every class must have a unique name. This is expressed as the simple REFACOLA constraint rule (put into the context of a language definition in Figure 4)

```
for all c1, c2 : class if c1 != c2 then c1.id != c2.id
```

which alone governs the RENAME CLASS refactoring. A similar condition applies to renaming features (members); however, here the situation is more complicated.

<pre> language Eiffel kinds DeclaredEntity <: ENTITY {id} Class <: DeclaredEntity Feature <: DeclaredEntity Local <: DeclaredEntity Renaming <: Feature {inheritedId} Reference <: REFERENCE {id} properties id : Identifier inheritedId : Identifier queries binds(Reference, DeclaredEntity) "Reference binds to DeclaredEntity" has(Class, DeclaredEntity) "Class defines or inherits Feature or Local" redefines(Feature, Feature) "1st Feature redefines 2nd" renames(Renaming, Feature) "Renaming renames inherited Feature, poss. default" merged(Renaming, Renaming) "Renamings are merged in immediate subclass" split(Renaming, Renaming) "Renamings are different feat. in immediate subclass" </pre>
<pre> ruleset Naming for Eiffel rules binding for all r : Reference, d : DeclaredEntity if binds(r, d) then r.id = d.id unique-class-identifier for all c1, c2 : Class if c1 != c2 then c1.id != c2.id unique-feature-identifier-1 for all c : Class, f1, f2 : Feature if has(c, f1) has(c, f2) f1 != f2 then f1.id != f2.id unique-feature-identifier-2 for all c : Class, f : Feature, l : Local if has(c, f) has(c, l) then f.id != l.id redefine-feature for all f1, f2 : Feature if redefines(f2, f1) then f2.id = f1.id rename-feature for all f : Feature, r : Renaming if renames(r, f) then r.inheritedId = f.id merging for all r1, r2 : Renaming if merged(r1, r2) then r1.id = r2.id splitting for all r1, r2 : Renaming if split(r1, r2) then r1.id != r2.id </pre>
<pre> refactoring RenameFeature for Eiffel uses Naming forced changes id of DeclaredEntity allowed changes id of DeclaredEntity values {old, fresh} id of Reference penalties changing-identifiers for all d : DeclaredEntity penalize d.id != old d.id with 1 inserting-rename-clauses for all r : Renaming penalize r.id != r.inheritedId with 2 </pre>

Fig. 4. REFACTOLA modules for the RENAME FEATURE refactoring for Eiffel

In Eiffel, a class can *define* a feature, *inherit* it from a superclass, *redefine* an inherited feature (corresponding to overriding), *rename* it (can be combined with redefining), or *undefine* it (i.e., make it abstract). Since Eiffel has no overloading, the names of features available in a class must be unique within that class, and also different from all local variables and formal parameters (together referred to as locals) of that class. In particular, renaming is mandatory if a class inherits two features of the same name from different superclasses, or one that has the same name as one of its own features. The problem is illustrated by the sample program

```

class A feature i : ANY end
class B inherit A feature j : ANY end
class C inherit A rename i as j end end

```

for which renaming feature *i* of class *A* to “*j*” (as a refactoring) must either be rejected (because subclass *B* already has a feature of the same name), or *B.j* must be given a fresh name, or a rename clause must be introduced in *B* which renames the inherited feature. Note that, that *A.i* is renamed to “*j*” in *C* (by a corresponding rename clause) does not constrain the refactoring, even if this means that after the refactoring, the feature is renamed to the same name — instead, in the above example, rather than updating the reference *i* in the rename clause of *C* to *j*, the rename clause can be dropped, giving us the refactored program

```

class A feature j : ANY end
class B inherit A rename j as i end feature j : ANY end
class C inherit A end

```

Alternatively, as mentioned above, feature *B.j* could have been renamed by the refactoring, saving the renaming of the inherited *j*.

The REFACOLA definitions for a RENAME refactoring that incorporates enough of the Eiffel specification to handle all options are shown in Figure 4. Except for the handling of rename clauses and the special treatment of repeated inheritance [9 §8.16.2], all definitions are straightforward.

To handle rename clauses as described above, we introduce a special kind of feature, called *Renaming*, and assume default renamings (defaulting to the same name; cf. Section 4.4) for all inherited features that are not explicitly renamed. This implies that the query *binds(r, d)* must bind *r* to a default renaming if the target of *r* inherits the feature without explicitly renaming it. *Renaming* declares an additional property, *inheritedID*, for the identifier of the feature it renames (which may itself be a renaming, default or proper); this is used to keep track of the identifier of that feature (the *inherited identifier* [6 §8.6.18]), which is necessary for determining the default status of a renaming upon writing back the solution of the constraint system (only renamings *r* for which *r.id* ≠ *r.inheritedID* translate to rename clauses in the program; cf. Section 4.4), and also for expressing the penalty inserting-rename-clauses, suggesting that renamings should not needlessly be introduced by the solver (cf. Section 4.3).

In case of repeated inheritance, i.e., inheritance of a feature from two immediate superclasses that have in turn inherited the feature from a common ancestor (the “diamond problem” of multiple inheritance), Eiffel merges the two inherited features into one, unless one or both are renamed in the superclasses so that their names differ — in that case, the subclass inherits two (different) features (the Repeated Inheritance rule; [9 §8.16.2]). Renaming of repeatedly inherited features may thus cause the splitting of a feature into two, or the merging of two features into one, which both likely affect the meaning of the program. Therefore, renaming must maintain the merging status of repeatedly inherited features, which is guaranteed by the rules named *merging* and *splitting*.

6.2 The CHANGE ACCESSIBILITY Refactoring

The classes of an Eiffel program do not only all exist in the same global namespace, they are also accessible (“available” in Eiffel jargon) by all other classes of the same program. The accessibility of the features of a class is controlled by an explicit export

mechanism: each feature definition can have a set of classes attached from whose bodies (including the bodies of the classes' subclasses) the feature is accessible. By way of re-exporting using export clauses [6 §8.7], subclasses may change the export status of their inherited features, which includes making them accessible by fewer (including no) classes. This accounts for one half of so-called *catcalls* (where “cat” stands for “change availability or type” [19]), which threaten subtyping.³ A refactoring should therefore not introduce catcalls.

The following program illustrates some of the constraints controlling accessibility:

```

class A feature {C, D} m do ... end end
class B inherit A end
class C feature m local a : A do ... a.m ... end end
class D feature m local b : B do ... b.m ... end end

```

In this program, accessibility of A.m could be reduced to include just C, but the access of m on an instance of B from class D would require introduction of an export {D} m clause in class B, as in

```

class A feature {C} m do ... end end
class B inherit A export {D} m end end

```

This however is insufficient, since making B.m inaccessible from C (the semantics of the export clause is not additive) makes the call a.m in C a catcall. The export clause in B must therefore include C as well.

The definitions for a CHANGE ACCESSIBILITY refactoring for Eiffel are shown in Figure 5.⁴ They are explained as follows:

- While an Eiffel class may introduce one feature clause or export clause per feature defined or re-exported, both a feature clause and an export clause can list several features that then all share the same export status. To express this, feature clauses and export clauses (jointly represented by program elements of kind Export) also have an accessibility property, and the accessibility of features is inferred from that of their exports by constraining them to equal the accessibility of their clause (the accessibility-inference rule). Note that, were the export clause to which a feature belongs subject to change by a refactoring, this constraint would require indirection, i.e., $f.\text{accessibility} = f.\text{export.accessibility}$ (see Section 4.1), with export being a property of features with program-dependent domain [Export] (see Section 3.2).
- The accessibility property is set valued. This is reflected in the accessible-feature rule, which requires that the location of a reference to a feature, which is (the body of) a class, is a subclass of at least one class in accessibility, the set of classes the feature is exported to. The rule consequent therefore involves an existentially quantified constraint (cf. [27]); its semantics is not detailed here since for technical reasons, in our implementation we will have to replace set-valuedness of accessibility with single-valuedness (see Section 7).

³ While [19] defines catcalls statically, the Eiffel compiler [7] inserts runtime type checks signalling exceptions on the occurrence of catcalls, so that introducing catcalls does not lead to uncompileable code, but may lead to new runtime errors and therefore may change behaviour.

⁴ Note that we use different language and ruleset modules here only to make each refactoring specification self-contained. In practice, modules would be shared between refactorings.

```

language Eiffel
kinds
  Class <: ENTITY
  Export <: DeclaredEntity {accessibility}
  Feature <: DeclaredEntity {accessibility}
  Reference <: REFERENCE {location}
properties
  accessibility : SetOf(Type)
  location : Type
domains
  Type = [Class] ordered by inherits
queries
  feature(Export, Feature) "Feature is a feature of the clause Export"
  accesses(Reference, DeclaredEntity) "Ref. accesses DeclEnt. from other class"
  requires(Feature, Feature) "1st Feature references 2nd Feature in its precondition."
  reexports(Feature, Feature) "1st Feature redefines export status of 2nd"
  inherits(Class, Class) "1st Class inherits from 2nd"

ruleset Exporting for Eiffel
rules
  accessibility-inference
    for all e : Export, f : Feature
    if feature(e, f)
    then f.accessibility = e.accessibility
  accessible-feature
    for all r : Reference f : Feature
    if accesses(r, f)
    then exists c : Class ([c] in f.accessibility and r.location <= [c])
  precondition-export
    for all r : Reference, f1, f2 : Feature
    if accesses(r, f1) requires(f1, f2)
    then exists c : Class ([c] in f2.accessibility and r.location <= [c])
  no-catcalls
    for all r : Reference, f1, f2 : Feature
    if accesses(r, f1) reexports(f2, f1)
    then exists c : Class ([c] in f2.accessibility and r.location <= [c])

refactoring ChangeAccessibility for Eiffel uses Exporting
forced changes accessibility of Feature
allowed changes accessibility of Feature, accessibility of Export

```

Fig. 5. REFACOLA modules for the CHANGE ACCESSIBILITY refactoring

- It is a prerequisite of design by contract that all clients of a feature are able to check that the preconditions of the feature are satisfied (by querying the features that are used in the precondition) [19, 6 §8.9.5]. This is expressed in precondition-export, which requires that each feature referenced from the precondition of another feature of the same class is accessible to at least the clients of the referencing feature.
- To avoid the introduction of catcalls to a program (cf. above), the no-catcalls rule requires that if a referenced feature is re-exported in a subclass, the re-exported feature of the subclass must also be accessible from the location of the reference.

6.3 The CHANGE DECLARED TYPE Refactoring

Using more general types in the declarations of variables is considered good practice, since it means depending on fewer features, thereby increasing decoupling (the Interface Segregation Principle [16]). However, generalizing the declared type of a variable is constrained by the use of that variable (which features are being accessed on it), and also, via assignment compatibility, by the type of the other variables it gets assigned to (whose types are constrained accordingly). Specializing a declared type is constrained similarly, the differences being that assignments propagate type change in the opposite direction and that the use of the type need not be checked: if the

program to be refactored is free of catcalls, features defined by a supertype that are available to clients are also available — to the same clients — from its subtypes.

The situation is more complicated, however, when the parameter types of redefining or redefined (the Eiffel terms for overriding and overridden) routines (methods) are to be changed: since Eiffel allows covariant redefinition of all parameters, a parameter type of a redefined routine may be generalized, and a parameter type of a redefining routine may be specialized, by a refactoring. However, since covariant redefinition may lead to catcalls (here the call of a routine with a parameter that may not be accepted by a redefining routine in a subclass, threatening static subtyping), care must be taken that no CHANGE DECLARED TYPE refactoring introduces a catcall.

Changing a declared type is different from the previous refactorings in that it changes static binding of references to features, which could affect the meaning of a program. However, in Eiffel (unlike in Java for instance) the binding changes that may be caused by a change of declared type depend on the type of the features' targets (aka receivers) alone (recall that Eiffel has no overloading). Since in Eiffel (again unlike in Java), all feature accesses (including field accesses) are dynamically dispatched, a change of receiver or parameter type never leads to a change of dynamic binding, so that except for creating new objects (instantiation; see below), maintaining (static) type correctness is enough to preserve binding.

The following short sample program illustrates some of the challenges posed by the CHANGE DECLARED TYPE refactoring in Eiffel:

```

class A end
class B inherit A feature n do end end
deferred class C feature m(b1 : B) deferred end end
class D inherit C redefine m end feature m(b2 : B) do b2.n end end
class CLIENT feature m local d:D; b3:B do create(b3); d.m(b3) end end

```

Here, type changes are constrained as follows:

- $b1 : B$ can be generalized to $b1 : A$, and $d : D$ can be generalized to $d : C$, but not both at the same time, since this would make $d.m(b3)$ in class CLIENT a catcall statically (cf. Footnote 3);
- $b2 : B$ cannot be generalized to $b2 : A$ since A does not define feature n ; and
- $b3 : B$ cannot be generalized to $b3 : A$ since this would require either generalizing $b2$ or both d and $b1$, where both alternatives, as we have just seen, are prohibited.

As for specialization,

- $b1 : B$ cannot be specialized without specializing $b2$ with it;
- $b2 : B$ cannot be specialized without specializing $b3$ with it; and
- $b3 : B$ can be specialized to $b3 : D$, but not to $b3 : C$, since this would make the instantiation $create(b3)$ illegal (instantiation of a deferred type).

The definitions necessary for the CHANGE DECLARED TYPE refactoring are shown in Figure 6. They are explained as follows:

- Assignment is the usual subtyping constraint covering assignment compatibility: it requires that the type of the right-hand side of the assignment is a subtype of the type of the left-hand side.
- Feature-defined constrains the type of the receiver (called *target* in Eiffel parlance) of a feature access to descendants of the root definition of that feature (called its *seed*; including the seed itself). Thus, the access is constrained to bind to a *version* ([6 §8.16.8]) of the feature (where versions are defined as having the same seed).

<pre> language Eiffel kinds DeclaredEntity <: ENTITY {id} Variable <: DeclaredEntity {type, location} Feature <: Variable Formal <: Variable Reference <: REFERENCE {id, type, location} Class <: DeclaredEntity properties id : Identifier type : Type location : Type domains Type = [Class] ordered by inherits queries assignment(Reference, Reference) "2nd Reference is assigned to the 1st" binds(Reference, DeclaredEntity) "Reference binds to DeclaredEntity" target(Reference, Reference) "2nd Reference is the target (receiver) of 1st" seed(Feature, Feature) "2nd Feature is seed (root definition) of 1st Feature" defines(Class, Feature) "Feature is defined, redefined, or renamed in Class" actual(Reference, Reference, Index) "1st Ref. is actual of 2nd Ref. at Index" formal(Formal, Feature, Index) "Formal is formal of Feature at Index" redefines(Feature, Feature) "2nd Feature redefines 1st" create(Reference) "object of Reference's type is created and assigned to it" current(Reference) "Reference is Current" result(Reference, Feature) "Reference is Result of query Feature" like(Variable, Variable) "1st Variable declared to have the type of 2nd" inherits(Class, Class) "1st Class inherits from 2nd" </pre>
<pre> ruleset Typing for Eiffel rules assignment for all l, r : Reference if assignment(l, r) then r.type <= l.type feature-defined for all r, t : Reference f, s : Feature if binds(r, f) target(r, t) seed(f, s) then t.type <= s.location version-binding for all r, t : Reference f1, f2, s : Feature c : Class if binds(r, f1) target(r, t) seed(f1, s) seed(f2, s) defines(c, f2) then t.type = [c] implies (r.id = f2.id and r.type = f2.type) accessible-feature for all r, t : Reference f1, f2, s : Feature c : Class if binds(r, f1) target(r, t) seed(f1, s) seed(f2, s) defines(c, f2) then t.type = [c] implies exists c' : Class ([c'] in f2.accessibility and r.location <= [c']) routine-call for all r,t,a : Reference r1,r2,s : Feature c : Class f : Formal i:Index if binds(r, r1) target(r, t) seed(r1, s) seed(r2, s) defines(c, r2) actual(a, r, i) formal(f, r2, i) then t.type = [c] implies a.type <= f.type instantiation for all r : Reference if create(r) then r.type = old r.type covariant-feature-redefinition for all f1, f2 : Feature if redefines(f2, f1) then f2.type <= f1.type covariant-parameter-redefinition for all r1, r2 : Feature f1, f2 : Formal i : Index if redefines(r2, r1) formal(f1, r1, i) formal(f2, r2, i) then f2.type <= f1.type no-catcalls for all r : Reference r1, r2 : Feature f1, f2 : Formal i : Index if binds(r, f1) redefines(r2, r1) formal(f1, r1, i) formal(f2, r2, i) then f2.type = f1.type type-of-Current for all r : Reference if current(r) then r.type = r.location type-of-Result for all r : Reference f : Feature if result(r, f) then r.type = f.type anchored-type for all v1, v2 : variable if like(v1, v2) then v1.type = v2.type </pre>
<pre> refactoring ChangedDeclaredType for Eiffel uses Typing forced changes type of Variable allowed changes type of Variable, type of Reference, id of Reference </pre>

Fig. 6. REFCOLA modules for the CHANGE DECLARED TYPE refactoring

- **Version-binding** constrains the identifier and type of a reference to equal those of the version of the feature the reference binds to, which depends on the type of the target. Since this type may be changed by the refactoring, this rule compiles a table of conditional constraints (one constraint per version of the feature) of which only one will be active per solution (which one depending on the value of `t.type`).
- **Accessible-feature** makes sure that a feature accessed from a changed receiver type is still accessible; it works analogously to **version-binding** (see below for a discussion).
- **Routine-call** constrains the types of the actual parameters of a routine call to subtypes of their corresponding formal parameter types (analogously to the assignment rule). The variable `i` of type `Index` is implicitly constrained to the number of parameters of the routine.
- Since in Eiffel the class that is instantiated depends on the type of the variable used in the instantiation, and since replacing an object with one of a different class potentially affects behaviour, the instantiation rule prevents the change of declared type of variables used in instantiations. This rule could be relaxed to allow changes to subtypes, if existing contracts guaranteed behavioural subtyping; however, a refactoring tool has no way of checking such contracts. Also, this would require that the class defining the subtype is not deferred (abstract; but note how, if the refactoring definition is extended to include the **Deferring** ruleset of Figure 2, it could set `deferred` to `false`, if no deferred feature of the class prevents this).
- **Covariant-feature-redefinition** and **covariant-parameter-redefinition** constrain all types involved in a feature redefinition (in Eiffel, this includes fields) to subtypes.
- **No-catcalls** requires that redefinition of a referenced routine (method) leaves parameter types unchanged.
- **Type-of-Current** and **type-of-Result** constrain the types of the special references representing the current instance (this in Java et al.) and the return value of a function.
- **Anchored-type** constrains an anchored type to equal the type it is anchored to [6 §8.11.17].

Note that one half of the **version-binding** rule constrains the identifiers of a reference and the feature that is being referenced, and thus really belongs to the **Naming** ruleset of Figure 4. In fact, ignoring the type constraints generated by **version-binding**, it is a generalization of the binding rule of the **Naming** ruleset, namely one that takes the variability of the type of the reference's target into account (which could be ignored for **RENAME FEATURE**, since this does not change types). Indeed, as can easily be seen, if `t.type` is constant, which of the if-parts `t.type = [c]` of the conditional constraints generated by **version-binding** will be satisfied is already known when the rule is applied (namely for that application for which `[c]` equals the current type of the target) — for all that are not (all but a single one), the constraint can be dropped, since it does not constrain the solution (a conditional constraint with a false precedent is always satisfied). **Version-binding** of Figure 6 thus collapses to the (unconditional) binding rule of Figure 4. Note that the `ChangeDeclaredType` definition allows the change of identifiers of references; this enables the refactoring in cases in which different versions of a feature have different names (so that references must be updated).

Identifiers are not the only “foreign” property to be handled by CHANGE DECLARED TYPE — by changing the type of a target, a reference using that target may re-bind to a version of the feature that is inaccessible for the client holding the reference. While accessibility would be covered by the accessible-feature rule of the Exporting ruleset of Figure 5, this rule, like binding of the Naming ruleset, assumes that the (static) binding of a reference to a feature is not changed by a refactoring, which is not the case for CHANGE DECLARED TYPE. Instead, the accessible-feature rule of the Exporting ruleset (Figure 5) must be generalized to that of Typing (Figure 6) which, analogously to the version-binding rule, introduces a table of constraints.

6.4 Combining Several Refactorings into One

While for maximum applicability both identifiers and accessibilities must be handled by CHANGE DECLARED TYPE, changing them is not part of the primary refactoring intent. On the other hand, RENAME FEATURE, CHANGE ACCESSIBILITY, and CHANGE DECLARED TYPE are naturally combined into a single CHANGE FEATURE DECLARATION refactoring allowing the intended change of identifier, accessibility, and one or more parameter types of a routine, all in one step. For this to work correctly, the effect of one constituent refactoring must be immediately visible by the others, which is achieved by letting the constraints generated by each ruleset share those constraint variables that represent same properties of same program elements. Also, a property assumed as variable in one ruleset must not be assumed as constant in another (cf. the above adaptations necessitated by changeability of declared type). Both is achieved by merging the language definitions (cf. Section 5.1) underlying the different ruleset and refactoring definitions into one.

Generally, combining the specifications of different elementary refactorings can increase the applicability of each individual refactoring, if the refactoring is allowed to make additional changes controlled by the other refactoring specifications. For instance, given the program

```
class A feature {ANY} f do end end
class B inherit A export {C} f end end
class C feature g local a : A; b : B do create b; a := b; a.f end end
```

using the ruleset of CHANGE ACCESSIBILITY alone it is not possible to change accessibility of the field A.f to {NONE}. By combining the rulesets of CHANGE ACCESSIBILITY and CHANGE DECLARED TYPE, however, this presents no problem: since the rules of the Typing ruleset (including the adapted accessible-feature rule) allow it that the declared type of the local a in class C is changed to B, and since there are no calls of f on a supertype of A in this program that could trigger Exporting rule no-catcalls on A.f, the refactoring can be performed (and the constraint solver finds the solution). However, changing the declared type of a local such as a in the above example is normally *not* associated with a CHANGE ACCESSIBILITY refactoring (or a CHANGE FEATURE DECLARATION refactoring for that matter); in practice, such a *generalized refactoring*, which blurs the boundaries between individual refactorings, would require some advanced interaction facilities with the user (who might not understand why the program is refactored the way it is), including an explanation component such as [13].

7 Experimental Results

To demonstrate the adequacy of REFACTOLA and the above refactoring specifications, and also to measure the efficacy of our *GenerateConstraints* algorithm, we have systematically (i.e., in batch mode) applied each refactoring to all suitable program elements (as specified in the refactoring definitions) of several Eiffel programs. The programs and indicators of their size are listed in Table 1; all programs are taken from the Gobo Eiffel Tools and Libraries [10], and are subsystems of EiffelStudio [7].

The refactorings were applied as follows. `RENAME FEATURE` was applied to all feature definitions with “other” as the target name. This name is frequently used in the benchmark programs (160 times), thus causing a fair number of name conflicts. `CHANGE ACCESSIBILITY` was also applied to all feature definitions, with target `NONE` (corresponding to `private`). `CHANGE DECLARED TYPE` was applied to features whose declared type was defined in the same project; targets were all immediate supertypes of the type, if in the same project.

The results of applying the individual refactorings to all programs of Table 1 are shown in Table 2. The numbers were obtained making a couple of restrictions:

- Since renaming is largely constrained by inequality constraints and since the refactoring definition of Figure 4 restricts the domains for identifiers a refactoring may change to two elements (the old and a fresh name), the solution space of renaming is of the order $O(2^n)$ (where n the number of features potentially having a name conflict), which is a problem for finding the best solutions if n is large. Therefore, we extended the *GenerateConstraints* algorithm of Figure 1 with an early evaluation step, pruning constraint generation for constraints known to be always satisfied, which is inherently the case for constraints of the kind `e1.id != e2.id`, if the domains of `e1.id` and `e2.id` are disjoint by construction (cf. Figure 4).
- Because of a limitation of our used constraint solver (Cream [28]), which does not support set-valued domains, we changed the domain of the accessibility property to `[Class]`, and adapted the constraint rules accordingly. That this presents no threat to the validity of our results can be seen by observing that only 8 (amounting to 1.2%) of all explicit exports in our sample projects listed more than one class.
- The times for generating the constraint system include querying, but not creating or searching the AST (which was first transformed to a database representation).

The results of Table 2 are interpreted as follows. `RENAME FEATURE` to “other” was of course always possible (since the domain of each identifier contained a fresh name; see the refactoring definition in Figure 4); that it is also extremely fast is due to the

Table 1. Sample programs and their sizes

Program	SLOC	Classes	Features	Program	SLOC	Classes	Features
Gobo Argument	2293	13	158	Gobo Pattern	240	6	11
Gobo Kernel	22913	143	1201	Gobo Regexp	8339	29	378
Gobo Lexical	22139	264	1168	Gobo String	73785	118	2270
Gobo Math	5233	13	331	Gobo Time	5664	34	369
Gobo Parse	18496	69	824	Gobo Utility	7710	46	543
				total	166812	735	7253

Table 2. Results of applying the generated refactorings to the programs of Table 1

REF.	AP. [§]	METRIC	AVG	MIN	¼ Q	MED	¾ Q	MAX
RENAME FEATURE	7253 /7253 (100%)	# of Constr. Variables	8	1	2	3	5	1002
		# of Generated Constr.	378	1	67	137	583	2115
		# of Generab. Constr. [†]	1232598	145331	483937	922843	2387543	2387543
		Reduction by (log ₁₀)	3.7	2.4	3.3	3.6	4.2	5.7
		Time Generating [ms] [§]	336	<.5	125	328	484	984
		Time Solving [ms]	2	<.5	<.5	<.5	<.5	234
CHANGE ACCESSIBIL.	5755/7253 (79%)	# of Constr. Variables	7	1	1	1	6	100
		# of Generated Constr.	26	1	3	5	23	1002
		# of Generable Constr.	30781	398	15926	28701	55089	55089
		Reduction by (log ₁₀)	3.4	1.2	2.8	3.6	4.3	4.7
		Time Generating [ms]	1	<.5	<.5	<.5	<.5	219
		Time Solving [ms]	54	<.5	<.5	<.5	<.5	19656
CHANGE DECL. TYPE	3198/9316 (34%)	# of Constr. Variables	9	1	3	4	8	892
		# of Generated Constr.	9	1	2	3	7	1016
		# of Generable Constr.	90150	823	14494	121763	121763	121763
		Reduction by (log ₁₀)	4.2	1.1	3.7	4.4	4.6	5.1
		Time Generating [ms]	897	<.5	16	31	62	33296
		Time Solving [ms]	93	<.5	<.5	15	31	15609

[§] applicability: ratio of how often the refactoring could be executed to the number of opportunities for application

[†] that is, generated without using the *GenerateConstraints* algorithm of Figure 1 (cf. Section 4.2)

[§] time measured on a contemporary laptop (2GHz Intel Centrino Duo with 2GB RAM, running Windows 7 operating system)

early evaluation, which reduces the search space dramatically (generating only 8 constraint variables on average). Without it, searching for a best solution (with a minimum number of renamings; cf. Section 6.1) the solver timed out after 5 seconds in more than half of all cases. Note that the search for a best solution de facto fell prey to early evaluation, causing that no constraint variables are generated for identifier properties that need not change.

CHANGE ACCESSIBILITY to NONE was possible in 79% of all applications of the refactoring. This number may seem surprisingly high, but is explained by the fact that the sample programs are libraries that are not necessarily clients of themselves (so that many features are never referenced; note that this also explains the small numbers of generated constraints). The relatively long maximum solving time of almost 20 s was taken by an unsolvable CSP with 118 constraints; overall, however, there were only 13 applications (all unsolvable) for which the solver required more than 5 s. If the generated CSP was solvable, it had precisely one solution; this follows from the refactoring specification (cf. Figure 5).

CHANGE DECLARED TYPE to a supertype was possible in 34% of all cases, indicating that there is opportunity for generalization in the samples (and that having the refactoring at one's disposal is indeed worthwhile). Compared to the other refactorings, times for generating are long; this due to the relatively high number and complexity of the involved constraint rules (Figure 6). Still, total time remains within reasonable bounds most of the times (less than 1 s on average, only 5% of all

applications take longer than 5 s), and even the longest times (34 s for generating and solving) are tolerable. The number of generated solutions is 1.2 on average, peaking at 512; however, in only 3% of all applications, there were more than two solutions.

Across all three refactorings, the number of constraints generated using the *GenerateConstraints* algorithm of Figure 1 is dramatically smaller (between 1 and almost 6 orders of magnitude) than what would have been generated without it. Measuring the concomitant reduction in the number of solutions was of course infeasible (due to the exponential size of the solution space) and therefore cannot be reported on here.

To show that the combination of different refactorings can increase applicability of a standard refactoring as suggested in Section 6.4, we have applied CHANGE ACCESSIBILITY, extended with the rulesets of RENAME FEATURE and CHANGE DECLARED TYPE and allowing it to change identifiers and types also, to the sample programs of Table 1 (again with target accessibility NONE). This enabled the refactoring in an additional 71 cases, increasing its applicability by 2%.

8 Conclusion

We have argued that for refactoring tools for a given programming language to be correct, they need to incorporate relevant parts of the language's specification. Generalizing prior work on constraint-based refactoring, which has modelled the typing [30] or accessibility [26] rules of Java using constraints, we have devised a refactoring constraint language that, together with accompanying compiler and framework, allows us to generate refactoring tools directly from specifications that mirror the syntactic and semantic rules of the language. By applying generated refactoring tools for the Eiffel programming language to a number of sample programs, we have shown that, despite the generality and declarative nature of our approach, the tools are usable in practice. Another result is that by the combination of the constraint rules underlying different refactorings (representing different aspects of a programming language), we can increase applicability of each individual refactoring.

Acknowledgments. The authors are indebted to Andreas Thies and Erland Müller for their contributions to mastering the refactoring challenges. Emmanuel Stapf helped with the internals of the Eiffel compiler; Naoyuki Tamura pointed us to the *element* constraint.

This work has been made possible by the Deutsche Forschungsgemeinschaft (DFG) under grant STE 906/4-1.

References

1. Andreae, C., Noble, J., Markstrum, S., Millstein, T.D.: A framework for implementing plug-gable type systems. In: Proc. of OOPSLA, pp. 57–74 (2006)
2. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Trans. Knowl. 1(1), 146–166 (1989)
3. Daniel, B., Dig, D., Garcia, K., Marinov, D.: Automated testing of refactoring engines. In: Proc. of ESEC/SIGSOFT FSE, pp. 185–194 (2007)
4. Donovan, A., Kiezun, A., Tschantz, M.S., Ernst, M.D.: Converting Java programs to use generic libraries. In: Proc. of OOPSLA, pp. 15–34 (2004)

5. Eclipse, <http://www.eclipse.org>
6. ECMA Standard Eiffel: Analysis, Design and Programming Language ECMA-367, 2nd edn. (2006)
7. EiffelStudio, <http://www.eiffel.com/products/studio>
8. Fowler Refactoring, M.: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
9. Fuhrer, R., Tip, F., Kiezun, A., Dolby, J., Keller, M.: Efficiently Refactoring Java Applications to Use Generic Libraries. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 71–96. Springer, Heidelberg (2005)
10. Gobo Eiffel Tools and Libraries, <http://www.gobosoft.com>
11. WG Griswold Program Restructuring as an Aid to Software Maintenance PhD Dissertation, University of Washington (1992)
12. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, <http://java.sun.com/docs/books/jls/>
13. Junker, U.: QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In: Proc of AAI, pp. 167–172 (2004)
14. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in Java. In: Proc. of ICSE, pp. 431–440 (2008)
15. Kiezun, A., Ernst, M.D., Tip, F., Fuhrer, R.M.: Refactoring for parameterizing Java classes. In: Proc. of ICSE, pp. 437–446 (2007)
16. Martin, R.: Agile Software Development: Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs (2003)
17. Marriott, K., Stuckey Programming, P.J.: with Constraints. MIT Press, Cambridge (1998)
18. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. *J. Softw. Maint. Evol.* 17(4), 247–276 (2005)
19. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice Hall International, Englewood Cliffs (1997)
20. Meyers, S., Duby, C.K., Reiss, S.P.: Constraining the structure and style of object-oriented programs. In: Proc of PPCP, pp. 200–209 (1993)
21. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD Dissertation, University of Illinois at Urbana-Champaign (1992)
22. Schäfer, M., Ekman, T., de Moor, O.: Sound and extensible renaming for Java. In: Proc. of OOPSLA, pp. 277–294 (2008)
23. Schäfer, M., Verbaere, M., Ekman, T., de Moor, O.: Stepping stones over the refactoring rubicon. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 369–393. Springer, Heidelberg (2009)
24. Schäfer, M., Dolby, J., Sridharan, M., Torlak, E., Tip, F.: Correct refactoring of concurrent java code. In: D’Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 225–249. Springer, Heidelberg (2010)
25. Schäfer, M., de Moor, O.: Specifying and implementing refactorings. In: Proc. of OOPSLA, pp. 286–301 (2010)
26. Steimann, F., Thies, A.: From public to private to absent: Refactoring JAVA programs under constrained accessibility. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009)
27. Steimann, F., von Pilgrim, J.: Refactoring with foresight, <http://www.fe.u.de/ps/docs/Foresight.pdf>
28. Tamura Cream, N.: Class Library for Constraint Programming in Java, <http://bach.istc.kobe-u.ac.jp/cream/>

29. Tip, F., Kiezun, A., Bäumer, D.: Refactoring for generalization using type constraints. In: Proc. of OOPSLA, pp. 13–26 (2003)
30. Tip, F.: Refactoring using type constraints. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 1–17. Springer, Heidelberg (2007)
31. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. MIT Press, Cambridge (1989)
32. Verbaere, M., Ettinger, R., de Moor, O.: JunGL: a scripting language for refactoring. In: Proc. of ICSE, pp. 172–181 (2006)
33. Xpand, <http://wiki.eclipse.org/Xpand>
34. Xtext — Language Development Framework, <http://www.eclipse.org/Xtext>
35. A Type System Framework for Xtext, <http://eclipse-labs.org/p/xtext-typesystem>

Modal Object Diagrams

Shahar Maoz*, Jan Oliver Ringert**, and Bernhard Rumpe

Software Engineering
RWTH Aachen University, Germany
<http://www.se-rwth.de/>

Abstract. While object diagrams (ODs) are widely used as a means to document object-oriented systems, they are expressively weak, as they are limited to describe specific possible snapshots of the system at hand. In this paper we introduce *modal object diagrams* (MODs), which extend the classical OD language with positive/negative and example/invariant modalities. The extended language allows the designer to specify not only positive example models but also negative examples, ones that the system should not allow, positive invariants, ones that all system's snapshots should include, and negative invariants, ones that no system snapshot is allowed to include. Moreover, as a primary application of the extended language we provide a formal verification technique that decides whether a given class diagram satisfies (i.e., models) a multi-modal object diagrams specification. In case of a negative answer, the technique outputs relevant counterexample object models, as applicable. The verification is based on a reduction to Alloy. The ideas are implemented in a prototype Eclipse plug-in. Examples show the usefulness of the extended language in specifying structural requirements of object-oriented systems in an intuitive yet expressive way.

*“... in the real world there are only objects.
Classes exist only in our minds.”, Nierstrasz [24]*

1 Introduction

The language of object diagrams (ODs) is part of the UML standard and is supported by many academic and commercial software modeling tools (see, e.g., [7,14,23,26,32]). The semantics of an object diagram is simple: an object diagram describes a possible instantiation of the system under development – a single object model – a snapshot of the system's structure made of concrete object instances and the relations between them. However, while ODs are useful and intuitive means to present example instances of object-oriented systems in formal and semi-formal contexts, their expressive power is rather weak, and they have no additional usages beyond these simple presentations.

* S. Maoz acknowledges support from a postdoctoral Minerva Fellowship, funded by the German Federal Ministry for Education and Research.

** J.O. Ringert is supported by the DFG GK/1298 AlgoSyn.

In this paper we introduce *modal object diagrams* (MODs), which extend the classical object diagram language with positive/negative and example/invariant modalities. The extended language allows to mark an object model not only as a *positive example*, describing a snapshot that the system should allow, but also as a *negative example*, which the system should not allow, as a *positive invariant*, which must be part of every snapshot of the system, or as a *negative invariant*, which must not be part of any snapshot of the system. Thus, the language supports the definition of very simple and intuitive yet expressively powerful specifications for the structure of the system to be; multi-modal specifications made of positive and negative examples and invariants. The syntax and semantics of modal object diagrams is formally defined in Sect. 3.

Moreover, as a primary application of the extended language, we consider a setup where a set of modal object diagrams is used as a specification that the system's class diagram (CD) should meet. The syntax of CDs is made of classes and the relationships between them, including inheritance and various associations. The semantics of CDs is given in terms of sets of objects and the relations between them. Thus, to support this application, we provide a formal verification technique that checks whether a CD – as a model of a system's structure defined by engineers – indeed satisfies (i.e., models) a multi-modal object diagrams specification. Given a CD and a multi-modal object diagram specification, the technique checks whether all positive examples in the specification are included in the CD's semantics, whether all negative examples are not included in the CD's semantics, whether all positive invariants are part of every object model in the CD's semantics, and whether all negative invariants are not part of any object model in the CD's semantics. In case of a negative answer, the technique outputs relevant counterexample object models, as applicable.

The verification technique is based on a transformation to Alloy [15]. Unlike previous works that consider the use of Alloy for the analysis of CDs (e.g., [1,30]), the input for our transformation consists not only of a CD but also of an OD (or a set of ODs). Moreover, the transformation itself is different, as it follows a pragmatic approach: we are not suggesting a meta-model level framework for general transformations but instead focus on solving the concrete engineering problem we have at hand. The verification technique is described in Sect. 4.

In order to test and evaluate our work we have implemented and integrated it into a prototype Eclipse plug-in. The plug-in allows the engineer to edit MODs and CDs, and to verify a selected multi-modal MOD specification against a selected CD. Indeed, all examples shown in this paper have been verified by our plug-in. We describe the plug-in and the results of our experiments in Sect. 5.

The introduction of MODs and the associated verification technique suggest a stepwise design methodology. In early stages in the design process, MODs will most often be used by domain experts or system analysts, to describe possible snapshots of a system; in doing so, designers stipulate that the system should at least be able to exhibit the positive examples shown in the MODs. As the process matures, knowledge will become available about structures that should not be possible, so the initial set of positive example MODs could be refined with

negative examples. Finally, in later stages, the analysts will be confident enough to define positive and negative invariant MODs. The verification technique we provide would aid the engineers in checking that their design indeed meets the concrete requirements defined by the MODs. We discuss this design process further in Sect. 7.4.

Object-oriented design constraints, similar to MOD specifications, can also be defined using the Object Constraint Language (OCL) [25]. Thus, one may consider using OCL instead of MOD or defining some combination of the two. We discuss the relation between OCL and MOD in Sect. 7.3.

The paper is organized as follows. The next section presents motivating examples for the use of multi-modal object diagram specifications. Sect. 3 formally defines the MOD language. Sect. 4 describes the verification problem and the technique to solve it. Sect. 5 presents our prototype implementation. Two extensions to the MOD language are discussed in Sect. 6. A discussion of limitations and advantages, a comparison with OCL, the use of MOD in the design process, and future work directions are presented in Sect. 7. Sect. 8 discusses related work and Sect. 9 concludes.

2 Examples

We start off with simple examples for multi-modal object diagram specifications, as they may be used during the design phase of a system. The examples are described semi-formally. The required formal definitions are given in Sect. 3.

2.1 Example I

Fig. 1 shows a specification MS_1 made of five MODs, prepared by a business analyst for a transportation services company. The specification includes three positive examples and two negative examples. *mod1.1* describes a car with a driver. *mod1.2* describes a car with two drivers. *mod1.3* describes a bus with a driver who has a manager. *mod1.1*, *mod1.2*, and *mod1.3* are all positive examples. *mod1.4* describes a negative example: a driver and a bus, not connected. Finally, *mod1.5* describes another negative example: a lone driver.

Given the specification MS_1 , which was provided by the business analyst, the system's engineers have designed the class diagram cd_1 shown in Fig. 2. Note that the engineers have suggested to generalize **Car** and **Bus** using an abstract super class **Vehicle**. As this example is small, it is easy to see that $cd_1 \models MS_1$. The engineers have used our plug-in to verify this.

2.2 Example II

Following further investigation of the company's structure, the business analyst prepared a revised MOD specification MS_2 , as shown in Fig. 3. The revised specification is made of four new MODs: a positive invariant, two negative invariants, and a positive example.

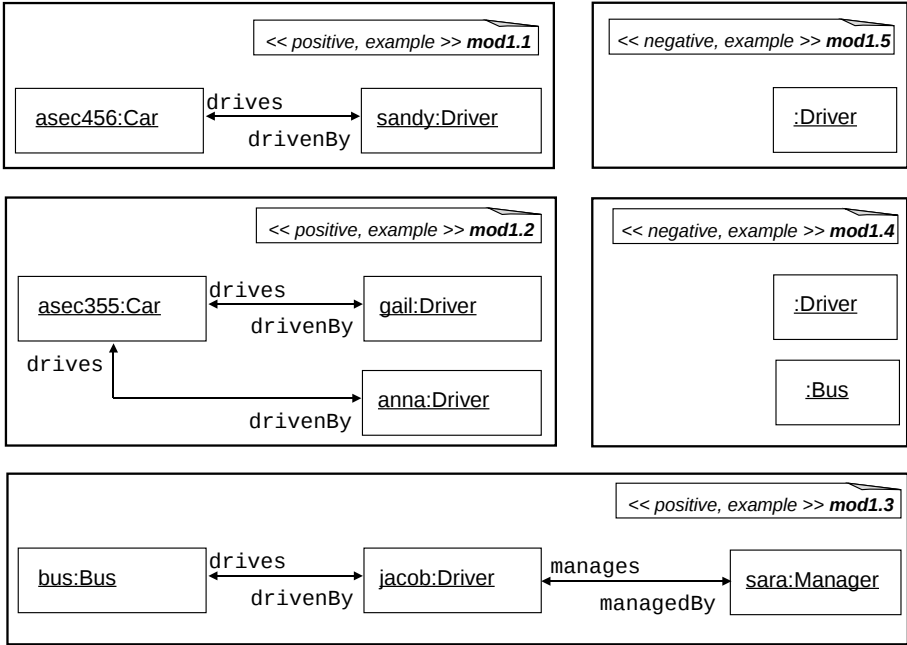


Fig. 1. The multi-modal MOD specification MS_1

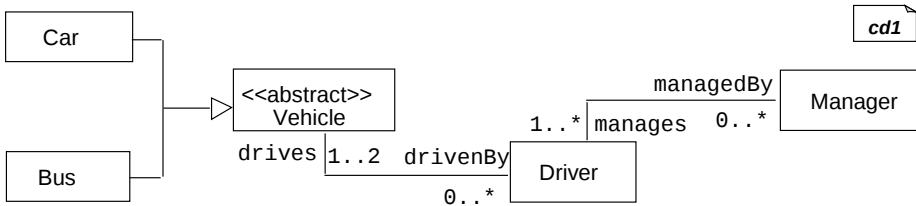


Fig. 2. cd_1 , a class diagram prepared for the specification MS_1

$mod2.1$ describes a positive invariant: every object model of the system must include at least one driver. $mod2.2$ describes a negative invariant: no object model of the system should include two managers. $mod2.3$ describes another negative invariant: a driver driving a bus, a car, and a sports car. Finally, $mod2.4$ describes a positive example: a manager managing an employee and a driver.

Given the MS_2 specification, the system’s engineers have designed the CD cd_2 shown in Fig. 4. In the new CD, the engineers added a class **Employee**, and defined **Driver** and **Manager** to be its sub classes. They also set the class **Manager** to be a singleton, to support the negative invariant defined in $mod2.2$.

Unfortunately though, using our plug-in the engineers have found that $cd_2 \not\models MS_2$. First, cd_2 requires that every driver will drive at least one vehicle, but

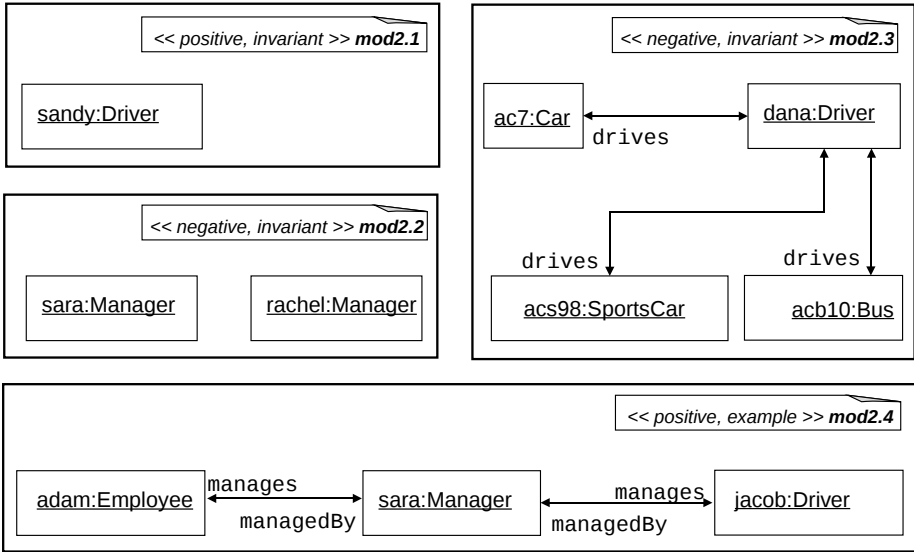


Fig. 3. The multi-modal MOD specification MS_2

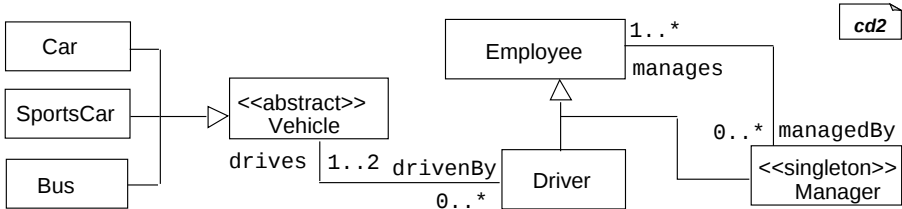


Fig. 4. cd_2 , a class diagram prepared for the specification MS_2 . Note, however, that $cd_2 \not\models MS_2$.

the driver in the positive example $mod2.4$ does not drive a vehicle. Second, cd_2 allows an object model consisting of a single manager that manages an employee that is not a driver. This contradicts the positive invariant $mod2.1$ (the related counterexample model was suggested by our plug-in). In addition, $cd_2 \not\models MS_1$. For example, $mod1.1$ shows a driver without a manager as a positive example, but according to cd_2 all object models should have exactly one manager (the same holds for $mod1.2$). Hence, the engineers should fix their CD or consult with the analyst on whether indeed a manager is required or not in every positive object model, and whether drivers should indeed drive at least one vehicle.

As the analysis progresses, the business analyst continues to learn about the system at hand and to provide the engineers with additional MODs. The engineers continue to design CDs that should meet the requirements set by the

analyst and use our plug-in to check those CDs against the MODs that the analyst provides.

The next section presents the required formal definitions for modal object diagrams. We return to the above examples in the latter parts of the paper.

3 Modal Object Diagrams

We give an overview of the CD and OD languages used in our work, and continue with formal definitions of MODs and the relation $cd \models MS$ between a class diagram cd and a multi-modal object diagram specification MS .

3.1 Class Diagrams and Object Diagrams

As a concrete CD language we use the class diagrams of UML/P [27], a conceptually refined and simplified variant of UML designed for low-level design and implementation. Our semantics of CDs is based on [34,8] and is given in terms of sets of objects and relationships between these objects. More formally, the semantics is defined using three parts: a precise definition of the syntactic domain, i.e., the syntax of the modeling language CD and its context conditions (we use MontiCore [18,23] for this); a semantic domain - for us, a subset of the System Model (see [34]) OM, consisting of all finite object models; and a mapping $sem : CD \rightarrow \mathcal{P}(OM)$, which relates each syntactically well-formed CD to a set of constructs in the semantic domain OM. For a thorough and formal account of the semantics see [4].

For example, the semantics of cd_1 shown in Fig. 2 includes all object models where all drivers drive one or two vehicles, all vehicles are driven by zero or more drivers, there are no vehicles that are not cars or buses but there may be cars and buses, every driver has zero or more managers, and every manager manages at least one driver. Note that the empty object model, which is an object model with no objects at all, is in the semantics of cd_1 . In addition, note that the semantics of cd_1 consists of an infinite number of object models.

As another example, the semantics of cd_2 shown in Fig. 4 includes all object models where all drivers drive one or two vehicles, all vehicles are driven by zero or more drivers, there are no vehicles that are not cars, buses, or sports cars, but there may be cars and buses and sports cars, every employee has zero or more managers, every driver is an employee, every manager is an employee, every manager manages at least one employee, and there is exactly one manager. The empty object model is not in the semantics of cd_2 because every object model in the semantics of cd_2 must include exactly one manager. Also, as in the semantics of cd_1 , the semantics of cd_2 consists of an infinite number of object models.

Note that we use a *complete* interpretation for CDs (see [27] ch. 3.4), roughly meaning that ‘whatever is not in the CD, should indeed not be present in the object model’. In particular, we assume that the list of attributes of each class is complete, e.g., a `driver` object with an `address` and a `salary` is not considered as part of the semantics of a `Driver` class with an `address` only. Also, the list of classes in the CD is considered complete, in the sense that its object models cannot include objects of classes not explicitly mentioned in the CD.

Also note that object names may be used in the OD for convenience, but they have no semantic meaning, i.e., the object name is not interpreted as an attribute **name**. Thus, e.g., an object **sara:Driver** has the same semantics as an object **dan:Driver** or an unnamed object **:Driver**.

The CD language constructs we support include generalization (inheritance), interface implementation, abstract and singleton classes, class attributes, uni- and bi-directional associations with multiplicities, enumerations, aggregation, and composition.

A note about notation: object diagrams refer to concrete syntactical expressions and object models refer to elements in the semantic domain. Still, the mapping from the abstract syntax to the semantic domain is in this case one-to-one, so we use OD and OM interchangeably. For example in the definition below we write $\forall pe \in MS.PE : pe \in sem(cd)$, although $MS.PE$ is a set of (modal) object diagrams while $sem(cd)$ is a set of object models. A more strict (yet inconvenient) notation should use $sem(pe) \in sem(cd)$.

3.2 Defining Modal Object Diagrams

We are now ready to present modal object diagrams, multi-modal object diagram specifications, and their relation to class diagrams.

Definition 1 (modal object diagram (MOD)). *A modal object diagram is a structure $mod = \langle od, p, q \rangle$ where $od \in OD$ is an object diagram, $p \in \{positive, negative\}$, and $q \in \{example, invariant\}$.*

Syntactically, we use stereotypes to denote the positive/negative and example/invariant modalities. Alternative syntactic means may be suggested, e.g., the use of dashed-line boxes in example MODs vs. solid-line boxes in invariant MODs.

Definition 2 (multi-modal object diagram specification). *A multi-modal object diagram specification is a set of MODs. Given a specification MS , the set of positive example MODs in MS is denoted $MS.PE$, the set of negative example MODs in MS is denoted $MS.NE$, the set of positive invariant MODs in MS is denoted $MS.PI$, and the set of negative invariant MODs in MS is denoted $MS.NI$. Any of these sets may be empty.*

We define the satisfaction relation between a CD and a multi-modal object diagram specification. Below we use $om_1 \subseteq om_2$ to note that all objects and links appearing in om_1 appear also in om_2 .

Definition 3 ($cd \models MS$). *A class diagram cd satisfies a multi-modal object diagram specification MS , denoted $cd \models MS$, iff*

1. $\forall pe \in MS.PE : pe \in sem(cd)$;
2. $\forall ne \in MS.NE : ne \notin sem(cd)$;
3. $\forall pi \in MS.PI, \forall om \in sem(cd) : pi \subseteq om$;
4. $\forall ni \in MS.NI, \forall om \in sem(cd) : ni \not\subseteq om$.

Note that the definition above uses a *complete*, rather than a *partial*, interpretation of positive example MODs. That is, it considers each OD to describe a complete object model rather than a partial one. We discuss an alternative partial semantics variant in Sect. 6.1.

Finally, since our verification technique, as described in the next section, is based on a transformation to Alloy [15], we need a bounded variant of the satisfaction relation. Below we use $|om|$ to note the maximal number of objects per class in om (objects of subclasses are counted also as objects of their super classes). Note that a bound needs to be applied only to invariants, because for example MODs, the MOD itself determines the size of the problem.

Definition 4 ($cd \models_k MS$). *A class diagram cd satisfies a multi-modal object diagram specification MS modulo a bound $k > 0$, denoted $cd \models_k MS$, iff*

1. $\forall pe \in MS.PE : pe \in sem(cd)$;
2. $\forall ne \in MS.NE : ne \notin sem(cd)$;
3. $\forall pi \in MS.PI, \forall om \in sem(cd)$ s.t. $|om| \leq k : pi \subseteq om$;
4. $\forall ni \in MS.NI, \forall om \in sem(cd)$ s.t. $|om| \leq k : ni \not\subseteq om$.

4 Verifying a CD against an MOD Specification

4.1 Problem Definition

The verification problem definition is as follows: given a multi-modal object diagram specification MS , a class diagram cd , and a bound k , check whether $cd \models_k MS$. Moreover, in case of a negative answer, provide relevant counterexample object models, as applicable for the negative and positive invariants at hand: for each unsatisfied $pi \in MS.PI$, provide $om \in sem(cd)$ s.t. $pi \not\subseteq om$; for each unsatisfied $ni \in MS.NI$, provide $om \in sem(cd)$ s.t. $ni \subseteq om$.

Our solution is based on a transformation to Alloy [15].

4.2 A Brief Overview of Alloy

Alloy is a textual modeling language based on relational first-order logic. An Alloy module consists of a number of signature declarations, fields, facts and predicates. The basic entities in Alloy are atoms. Each signature denotes a set of atoms. Each field belongs to a signature and represents a relation between two or more signatures. Such relations are interpreted as sets of tuples of atoms. Facts are statements that define constraints on the elements of the module. Predicates are parametrized constraints, which can be included in other predicates or facts.

Alloy Analyzer is a fully automated constraint solver for Alloy modules. The analysis is achieved by an automated translation of the module into a Boolean expression, which is analyzed by SAT solvers embedded within the Analyzer. The analysis is based on an exhaustive search for instances of the module. The search space is bounded by a user-specified scope, a positive integer that limits the number of atoms for each signature in an instance of the system that the solver analyzes.

The Analyzer checks for the validity of user-specified assertions. If an instance that violates the assertion is found within the scope, the assertion is not valid. If no instance is found, the assertion might be invalid in a larger scope. Used in the opposite way, one can look for instances of user-specified predicates. If the predicate is satisfiable within the given scope, the Analyzer finds an instance that proves it. If not, the predicate may be satisfiable in a larger scope. Sect. 7 discusses the advantages and limitations of using Alloy for our problem.

4.3 Solution by Transformation to Alloy

The transformation consists of three parts: handling the CD, handling each of the MODs, and generating of Alloy run commands. The complete transformation details are given in [19]. Here we give an overview of the transformation, using generated Alloy code taken from some of the examples shown earlier in Sect. 2.

Handling the CD. The input CD is transformed into a set of Alloy signatures, functions, and facts. Each class is transformed into an Alloy signature of a corresponding name, with fields defined according to the associations given in the CD. Alloy functions are defined in order to specify sets of objects of specific concrete classes, taking into account the information about inheritance hierarchy defined in the CD. Finally, Alloy facts are defined to express the types and multiplicities involved in the associations defined in the CD.

For example, the generated Alloy signatures, functions, and facts for the class diagram cd_2 of Fig. 4, are shown in Listings 1.1, 1.2, and 1.3.

Listings 1.1 shows the Alloy signatures for all the classes defined in the CD, with fields defined according to their associations. For example, see the `managedBy` field defined in line 2 for the signature `Employee`. The keyword `extends` is used to model class inheritance (see, e.g., lines 13-15, where the three sub classes of `Vehicle` are defined). Alloy's `one` keyword is used to model the singleton requirement specified by the `singleton` stereotype in the CD (see line 7). Finally, Alloy's `abstract` keyword is used to model the abstract requirement specified by the `abstract` keyword in the CD (see line 10).

Listings 1.2 shows the Alloy functions that specify sets of objects of concrete classes. Each function specifies a single set consisting of all objects of its corresponding class only, that is, without objects of its sub classes. For example, in line 1, the function `EmployeeOnly` is defined as the set consisting of all employees that are not drivers or managers.

Listings 1.3 shows the types and multiplicities of the associations defined in cd_2 . The `VehicleIsAbstract` fact specifies that the set `VehicleOnly` includes no elements. The symmetry of the bi-directional association between `Driver` and `Vehicle` is specified by requiring that the restriction of `Driver` to the field `drives` (as a relation) is the inverse of the restriction of `Vehicle` to the field `drivenBy`. Note that by definition this applies to all the sub classes of `Vehicle` too. The multiplicity constraints of the association between `Driver` and `Vehicle` are specified by limiting the size of the relevant sets referenced by the corresponding signatures' fields. The last two facts specify the symmetry and the

```

1 sig Employee {
2   managedBy: set Manager
3 }
4 sig Driver extends Employee {
5   drives: set Vehicle
6 }
7 one sig Manager extends Employee {
8   manages: set Employee
9 }
10 abstract sig Vehicle {
11   drivenBy: set Driver
12 }
13 sig Car extends Vehicle { }
14 sig Bus extends Vehicle { }
15 sig SportsCar extends Vehicle { }

```

Listing 1.1. Generated Alloy signatures for cd_2

```

1 fun EmployeeOnly: set univ {Employee-(Driver + Manager)}
2 fun DriverOnly: set univ {Driver}
3 fun ManagerOnly: set univ {Manager}
4 fun VehicleOnly: set univ {Vehicle-(Car+Bus+SportsCar)}
5 fun CarOnly: set univ {Car}
6 fun BusOnly: set univ {Bus}
7 fun SportsCarOnly: set univ {SportsCar}

```

Listing 1.2. Generated Alloy functions for cd_2

multiplicities constraints for the association between `Employee` and `Manager` in a similar way.

Handling Each of the MODs. Each input MOD is transformed into a predicate. The predicate consists of a conjunction of a number of parts. First, all objects in the diagram are listed (anonymous objects are given a random unique name). Second, the concrete types and number of occurrences is specified, making sure that super classes are not handled as sub classes. Finally, the links between the objects are specified using field assignments.

Listing 1.4 shows the generated predicate for the positive example MOD *mod2.4* of Fig. 3. Lines 3-4 list the names of the objects and their types by declaring corresponding variables. Lines 6-7 specify the concrete classes these objects belong to. Line 8 defines the number of objects of each class (e.g., if there were two or more managers, as in *mod2.2*, this would include the statement `# {sara, rachel} == 2`, to make sure that each variable references a distinct object). Line 10 specifies that the universe of objects for Alloy is exactly the set of objects listed in the MOD (this statement is omitted for invariant MODs, see below). Finally, lines 13-16 specify the concrete associations between the objects.

```

1 fact VehicleIsAbstract { # VehicleOnly == 0 }
2 fact Asso_Driver_drives_drivenBy_Vehicle_symmetry {
3   Driver <: drives = ~((Vehicle <: drivenBy))
4 }
5 fact Asso_Driver_drivenBy_drives_Vehicle_Mult {
6   all var: Driver | # var.drives >= 1 && # var.drives <= 2
7   all var: Vehicle | # var.drivenBy >= 0
8 }
9 fact Asso_Employee_managedBy_manages_Manager_symmetry {
10  Employee <: managedBy = ~((Manager <: manages))
11 }
12 fact Asso_Employee_manages_managedBy_Manager_Mult {
13   all var: Employee | # var.managedBy >= 0
14   all var: Manager | # var.manages >= 1
15 }

```

Listing 1.3. Generated Alloy facts for *cd2*.

```

1 pred checkFull {
2   // all objects in our OD
3   some adam: Employee | some jacob: Driver |
4     some sara: Manager |
5   // make sure a superclass is not handled as a subclass
6   adam in EmployeeOnly and jacob in DriverOnly
7   and sara in ManagerOnly
8   and # {adam} == 1 and # {jacob} == 1 and # {sara} == 1
9   // define universe
10  and univ = {adam + jacob + sara + Int}
11
12  // links between them
13  and sara.manages = {jacob + adam}
14  and jacob.managedBy = {sara}
15  and adam.managedBy = {sara}
16  and jacob.drives = none
17 }

```

Listing 1.4. Generated predicate for the positive example *mod2.4*. If this was an invariant (positive or negative), the predicate would have been named `checkPart` and the conjunct defining the universe (line 10) would have been omitted.

The difference between example MODs and invariant MODs (whether positive or negative) is manifested in the transformation as follows. For example MODs, the predicate includes an additional conjunct, which defines the universe for Alloy as exactly the set of objects listed in the diagram. For invariant MODs, in contrast, this conjunct is not added, as the universe for Alloy is allowed to include more objects.

As described above, Listing 1.4 shows the generated predicate for the positive example MOD *mod2.4* of Fig. 3. If the MOD was not an example but an


```

1 run checkFull for 6 but exactly 1 Driver ,
2   exactly 3 Employee , exactly 1 Manager

```

Listing 1.5. Generated Alloy run command for the positive example *mod2.4*

invariant MOD (positive or negative), the generated predicate would have been named `checkPart` instead of `checkFull` and would not have included the conjunct defining the universe. Lines 13-16 would also have changed, to reflect that additional links may exist.

Generating Run Commands for Alloy. Finally, we generate a set of run commands for Alloy, each corresponding to one of the MODs in the specification.

For positive or negative example MODs, the run command simply runs the corresponding generated predicate with exact per-class scopes taken from the MOD itself (the computation of the per-class scope takes super classes into consideration too). As an example, the run command for the positive example MOD *mod2.4* is given in Listing 1.5. The same run command is used to check for a negative example; the only difference between negative and positive examples is in the interpretation of the result that is provided by Alloy.

Checking for a positive invariant is done by asserting the generated `checkPart` predicate. That is, the generated command checks an Alloy assertion named `checkInvariant`, which includes the `checkPart` predicate. The scope for this check cannot be taken from the input MOD and is defined by the user. As an example, the check command for a positive invariant MOD, with a user-defined scope of 6, is given in Listing 1.6. Checking for a negative invariant is done using the `checkPart` predicate, with a user-defined scope.

5 Implementation and Evaluation

In order to test and evaluate the MOD language and the verification technique we have implemented a prototype Eclipse plug-in that allows the engineer to edit MODs and CDs and to verify a selected CD against an MOD specification. The prototype implementation, examples, and related materials are available from [22].

The plug-in includes a textual editor for MODs and CDs. The editor was generated using MontiCore [23] grammars (including a parser, syntax highlighting, outline view etc.), and the addition of the modalities to the ODS is done

```

1 assert checkInvariant {
2   checkPart
3 }
4
5 check checkInvariant for 6

```

Listing 1.6. Generated Alloy assert statement and check command for a positive invariant MOD, with a user defined scope of 6

using stereotypes. When the user selects a number of MODs and a CD, she can execute the verification. The transformation to Alloy is implemented using templates written in FreeMarker [9] and the execution of the generated module run commands is done using Alloy's APIs (the analysis is fully automated so the engineer does not need to see the generated Alloy code). Several parameters may be selected, e.g., the SAT solver and the scope that Alloy will use in the analysis.

The results of the verification process are shown in a hierarchical table (an Eclipse view), displaying which MODs are modeled by the CD and which ones are not. When the engineer clicks the name of an invariant MOD that is not modeled by the selected CD, if any, the plug-in displays a relevant generated counterexample OD in the main editor pane of the IDE.

5.1 Example Results

We have used the plug-in to examine several MOD specifications and related class diagrams, including the examples shown in Sect. 2. Fig. 5 shows a screen capture of the eclipse IDE, displaying several CD and MOD files on the explorer view on the left (files with extensions `.cd` and `.od`), the summary results of a verification process at the bottom of the screen, and a generated counterexample OD in the main view (an outline of the OD is shown on the right).

Specifically, the figure shows the results of verifying cd_2 against the four MODs of the multi-modal specification MS_2 (shown earlier in figures 4 and 3). As expected, the results table shows two failures and two passes. Indeed, $cd_2 \not\models \{mod2.1\}$ and $cd_2 \not\models \{mod2.4\}$. The counterexample shown in the main view relates to the positive invariant $mod2.1$: it shows an object diagram that consists of an employee, a manager, and two cars, where the manager manages herself and the other employee. This object model is in the semantics of cd_2 but it does not include a driver. Thus, it proves that the positive invariant $mod2.1$ is not satisfied by cd_2 .

As the example shows, the counterexample found by Alloy is not necessarily the smallest possible counterexample (within the user-specified scope). This points to a limitation in our technique. In the future, it may be worthwhile to develop a technique that produces minimal counterexamples according to some minimality criteria (e.g., number of classes, total number of objects, etc.).

5.2 Performance Results

Table 1 shows performance results from our experiments. Experiments were done using Alloy version 4.1.10 with SAT4J [28], on a regular laptop computer, Intel Dual Core CPU, 2.8 GHz, with 4 GB RAM, running Windows Vista. The CDs and MODs are the ones presented in Sect. 2.

For each CD and MOD, the table shows whether the verification passed or failed (i.e., whether the CD satisfies the MOD or not), some details on the SAT formula that Alloy generated (number of variables etc.), and the total time it took to run the verification (constructing the formula + solving it), in milliseconds. The column titled *Scope* reports on the scope used in the verification:

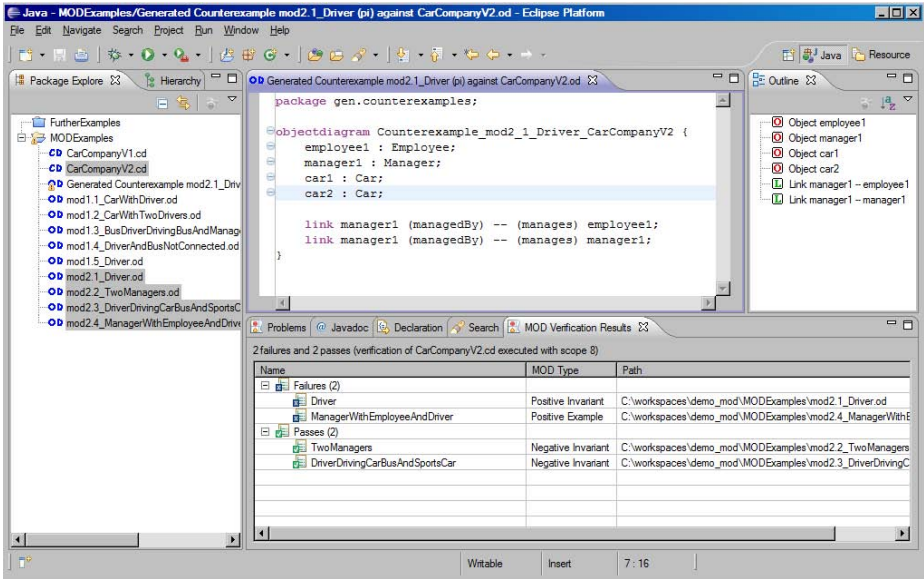


Fig. 5. A screen capture from Eclipse, displaying several CD and MOD files on the explorer view on the left (files with extensions `.cd` and `.od`), the summary results of a verification process at the bottom of the screen, and a generated counterexample OD in the main view (an outline of the OD is shown on the right). Specifically, the figure shows the results of verifying cd_2 against the four MODs of the multi-modal specification MS_2 , with two failures and two passes. The counterexample relates to the verification of the positive invariant $mod2.1$.

as explained in Sect. 4.3, for invariants MODs we use a user-defined scope; for example MODs, specific per-class scopes are taken from the MOD itself.

Interestingly, for some of the MODs, the generated Alloy formula was empty (had zero variables). That is, for these MODs, Alloy was able to determine the result without using the SAT solver. This happens when the generated Alloy module is very simple, e.g., when the `checkPart` predicate includes an immediate violation (contradiction) of one of the facts.

The verification of example MODs is, in general, simpler and faster, as it takes per-class scopes from the MOD at hand and its solution space is relatively small. The verification of invariant MODs, in contrast, is more complicated, and its solution space and performance depends on the user defined scope.

The performance results in Table 1 show that for relatively small models, MOD/CD verification runs very fast. However, we do have other examples, not shown here, where MOD verification uses many more variables and clauses and takes much more time to compute. Given these results, in the future, we plan to develop heuristics to improve the scalability of invariant MOD verification, using, e.g., abstraction / refinement techniques, decomposition for early detection of independent sub models, etc. See the short discussions in sections 7.1 and 7.2.

Table 1. Results from experimenting with the verification of example MODs and CDs

CD	MOD	Modalities	Scope	Result	Vars/primary vars/clauses	Time (ms)
cd_1	$mod1.1$	PE	by OD	pass	141 / 19 / 209	20 + 7
cd_1	$mod1.2$	PE	by OD	pass	371 / 34 / 581	18 + 8
cd_1	$mod1.3$	PE	by OD	pass	37 / 7 / 57	5 + 5
cd_1	$mod1.4$	NE	by OD	pass	132 / 19 / 192	6 + 0
cd_1	$mod1.5$	NE	by OD	pass	276 / 36 / 449	6 + 1
cd_2	$mod2.1$	PI	8	fail	2107 / 166 / 4360	17 + 10
cd_2	$mod2.2$	NI	8	pass	0 / 0 / 0	7 + 0
cd_2	$mod2.3$	NI	8	pass	2794 / 197 / 5867	17 + 7
cd_2	$mod2.4$	PE	by OD	fail	263 / 36 / 466	4 + 0
cd_2	$mod1.1$	PE	by OD	fail	0 / 0 / 0	5 + 0
cd_2	$mod1.2$	PE	by OD	fail	0 / 0 / 0	3 + 0
cd_2	$mod1.3$	PE	by OD	pass	49 / 9 / 71	4 + 11
cd_2	$mod1.4$	NE	by OD	pass	0 / 0 / 0	2 + 0
cd_2	$mod1.5$	NE	by OD	pass	0 / 0 / 0	4 + 0

6 Extensions

We present and discuss two extensions to the basic MOD language, inspired by [27]: partial vs. complete positive examples and parametrized ODs.

6.1 Partial vs. Complete Positive Examples

We distinguish between partial and complete positive examples. Roughly, a partial positive example object diagram specifies an object model that should be extensible to a positive example object model.

Recall the examples discussed in Sect. 2. There, we saw that $cd_2 \not\models MS_1$ because the positive examples $mod1.1$ and $mod1.2$ did not include a manager. Using the distinction between partial and complete positive examples, the analyst can specify that these MODs are partial positive examples and not complete ones. Doing so will make cd_2 satisfy MS_1 .

Syntactically, we specify that an OD is partial using a stereotype `partial`. The semantics for partial positive examples is formally defined as follows:

Definition 5 ($cd \models ppe$). *Given a class diagram cd and a partial positive example object diagram $ppe = \langle od, partial, positive, example \rangle$, we say that cd satisfies ppe , denoted $cd \models ppe$, iff $\exists od' \in sem(cd)$ s.t. $od \subseteq od'$.*

Updating Definitions 2, 3, and 4 from Sect. 3 to include partial positive examples is straightforward. The combination of `partial` with modalities other than positive examples is considered syntactically incorrect and its semantics is undefined (since invariant MODs already have a partial interpretation).

Note that the verification technique we described in Sect. 4 already supports MOD specifications with partial positive examples. Specifically, recall the `checkPart` predicate, which we use in the computation of invariants within an

assert statement. Running this predicate without an assert provides the required verification for partial positive examples.

6.2 Parametrized Object Diagrams

We extend the classical object diagram language with parameters, which may be used for attribute values or for object types. For example, instead of assigning a specific value to an attribute, the designer may assign it a parameter, and then define the set or range of values this parameter may take. Multiple parameters may be used in a single object diagram and the same parameter may appear more than once in a diagram. The semantics of parametric object diagrams is a natural extension of the classical semantics: it consists of the set of object models obtained by creating a set of non-parametric copies of the diagram, where in each copy different (combination of) values are assigned to the parameters.

For example, to specify that a driver's experience level can be either novice, regular, or expert, only a single (positive example) object diagram needs to be drawn, showing a driver whose `experience` attribute equals the OD parameter `level`, and $level \in \{novice, regular, expert\}$. Thus, the parametric extension allows the designer to create succinct object diagram specifications.

The combination of parametric object diagrams and modal object diagrams yields a powerful specification language. We give an example below.

Fig. 6 shows an MOD specification MS_3 , made of three parametric MODs. *mod3.1* is a negative invariant: it specifies that a driver cannot have an age lower than 16 (more formally, that any object model of the system should not include a driver whose age is between 1 and 15). *mod3.2* is a positive example: it specifies that a driver can drive a car, a sports car, or a bus (more formally, that a driver driving a car, a driver driving a sports car, and a driver driving a bus, are all positive examples of object models of the system). Finally, *mod3.3* is a negative example: it specifies that a driver who has novice or regular level of experience, is not allowed to drive a sports car that has medium or high engine power.

The verification technique we presented can be extended to support the parametric extension. The corresponding CDs may need to be enhanced with OCL constraints, which may get rather complicated, so the analysts and engineers should agree on the level of detail they want the specification and CDs to use.

7 Discussion and Future Work

We discuss advanced topics related to our work, its advantages and limitations, and related future work directions. These include a discussion of complexity and performance, the bounded scope limitation, the relationship between MOD and OCL, the use of MOD in the design process, and the problem of synthesizing a CD from an MOD specification.

7.1 Complexity and Performance

The transformation of the CD and MODs to Alloy is linear in the size of the input diagrams. It requires only a constant number of iterations over the diagrams' syntax and the construction of constant number of linear size 'symbol tables'.

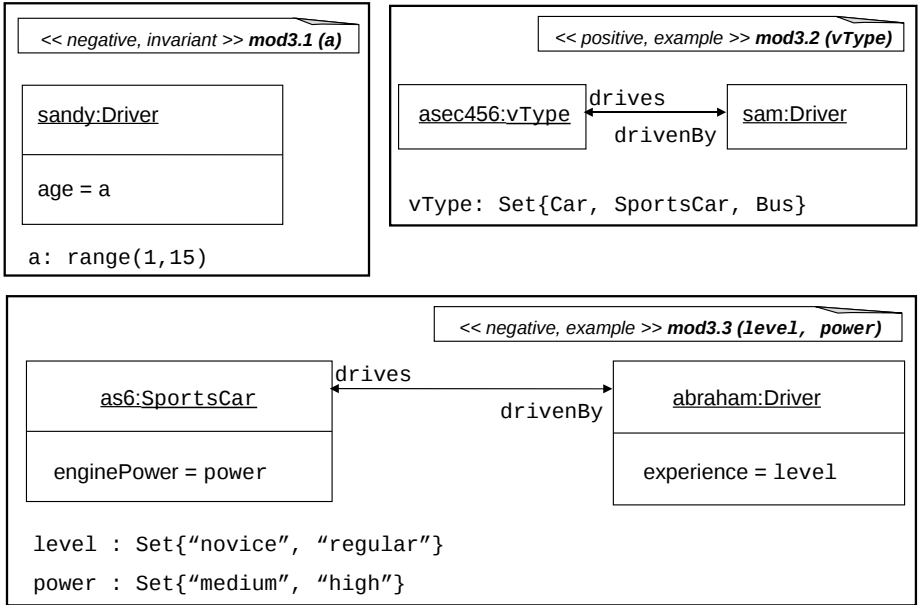


Fig. 6. The multi-modal parametrized MOD specification MS_3

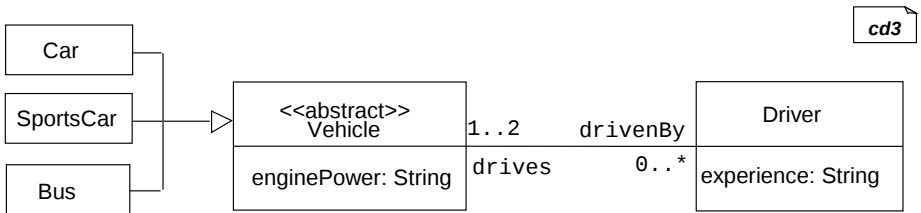


Fig. 7. A class diagram for the parametrized MOD specification MS_3

The computation by Alloy using a SAT solver, may be exponential in the size of the input diagrams. For example MODs, the solution space is relatively small and depends on the number of objects defined in the OD. In the case of invariant MODs, the solution space depends on the user-defined scope k . Although the two problems are rather different, we use a unified approach that solves both.

As discussed in Sect. 5, experience shows that our technique works very fast for relatively small models. Indeed, most works dealing with reasoning about CDs use rather small CDs in their experiments (see, e.g., [31,33]). Still, as future work, to make MOD verification practical for real-world projects, it would be necessary to develop heuristics that may accelerate the performance of Alloy in verifying larger MODs, experiment with the different SAT solvers supported by Alloy, or define a direct translation into SAT (as was suggested in [31]).

7.2 The Bounded Scope Limitation

The verification of positive and negative example object diagrams is sound and complete. The required scopes are taken from the example object diagrams themselves and so the answer is not only sound but complete.

The verification of positive and negative invariant object diagrams is however bounded by the user-defined scope. Specifically, it may be the case that for some cd and MS , $cd \not\models MS$ but $cd \models_k MS$ for some given k . A simple concrete example is as follows. Consider a CD cd_a consisting of two classes, C_1 and C_2 , and an association of multiplicity of exactly 1 to 5 between C_1 and C_2 . Assume a negative invariant object diagram ni consisting of two instances of C_1 and no instances of C_2 . Clearly, $cd_a \not\models \{ni\}$ but $cd_a \models_k \{ni\}$ for any $k < 10$.

The bounded version $cd \models_k MS$ is thus indeed strictly weaker than the general version $cd \models MS$. Our use of Alloy is sound and complete for the bounded version but is neither sound nor complete for the general version.

To conclude, the use of Alloy, and consequently the encoding of our verification problem as an instance of SAT, carries the significant price of bounded analysis. Nevertheless, we adapt the small scope hypothesis of [15] to our domain, and suggest that in many cases, although the models involved may be large, counterexamples for their unsatisfaction could be rather small.

As future work, heuristics may be developed to make automatic or semi-automatic informed guesses about suitable scopes that could reduce given problems into equivalent smaller ones where lower scopes are ‘good enough’, or to identify cases where one could automatically prove that a higher scope will not change the analysis results.

7.3 MOD and OCL

The Object Constraint Language (OCL) [25] is a declarative language for describing rules that apply to Unified Modeling Language (UML) models (and, more generally, to any Meta-Object Facility (MOF) meta-model). OCL is based on first-order predicate logic. As MODs specify constraints on object-oriented models too, discussing the relationship between MOD and OCL is worthwhile.

OCL is interpreted in the context of a UML diagram and is limited to specifying invariants, i.e., constraints that hold for all its instantiations. Thus, given that a CD context is provided, invariant MODs, positive and negative, can be specified using OCL. Moreover, negative example MODs can also be specified in OCL, by specifying a negative invariant that constrain also the set of all instances (the universe) to the set of existing instances listed in the MOD. Positive example MODs cannot be specified in OCL.

As a small example, recall *mod2.2*, which shows a negative invariant MOD. This MOD is semantically equivalent to the OCL code shown in Fig. 8 (to edit the OCL code that we show we used the Dresden OCL Eclipse plug-in [6]). If *mod2.2* was a negative example, the OCL code in Fig. 8 should have been extended, to specify, inside the outermost negative clause, that the total number of managers is two. Furthermore, every other class from the CD should be listed, to specify

```

package cd01
  context Manager
  inv inv01: not
    (Manager.allInstances()->
      exists( sara:Manager | Manager.allInstances()->
        exists( rachel:Manager |
          -- the two objects are distinct
          not(sara=rachel) and
          -- objects are of the specified types
          -- and not of any of their sub types
          sara.oclIsTypeOf(Manager) and
          rachel.oclIsTypeOf(Manager) and
          -- sara and rachel do not manage anyone
          sara.manages.asSet()->size()=0 and
          rachel.manages.asSet()->size()=0 and
          -- sara and rachel not managed by anyone
          sara.managedBy.asSet()->size()=0 and
          rachel.managedBy.asSet()->size()=0
        )
      )
    )
endpackage

```

Fig. 8. An OCL representation of MOD *mod2.2* from Fig. 3

that the size of its instances set is zero. For comparison purposes, Fig. 9 shows the textual representation of MOD *mod2.2* in our object diagram language (the language grammar is defined in MontiCore [18,23]).

These examples demonstrate that although it is impossible to specify positive example MODs in OCL, it is formally possible to specify invariant and negative example MODs using OCL. Yet, it is clearly very inconvenient, because manual writing of such OCL statements is obviously technically difficult and error prone. Thus, we chose to introduce MODs due to their readable and succinct representation, which makes them usable and attractive, using either textual or visual concrete syntax, not only for software engineers but also for non-SE specialists such as business analysts or other domain experts. On top of classical ODs, MODs make the notion of modality explicit; they integrate the intuitive concrete representation of the OD language with a limited set of predefined natural modalities.

Finally, as OCL is much richer than MOD in the kinds of invariants it can specify in the context of a given diagram, it may be interesting to follow [27] and define a combination of OD (MOD) and OCL. We discuss this combination in the related work section.

7.4 Using MODs in the Design Process

Just like classical ODs, MODs are simple and intuitive to define, since the addition of modalities does not change the basic syntax and semantics of


```

package test.examples;

<<negative, invariant>> objectdiagram TwoManagers {
    sara : Manager;
    rachel : Manager;
}

```

Fig. 9. The textual representation of MOD *mod2.2* from Fig. 3, as used in our work

describing a single concrete instance. In particular, ODs are much simpler than CDs, as they do not show inheritance and interface implementation relations. Moreover, CDs are made of abstract entities – classes – which do not ‘exist’ in the ‘real world’. As Oscar Nierstrasz puts it, “Classes exist only in our minds” [24]. ODs, in contrast, are made of concrete entities – objects – which indeed ‘exist’, both in the real world and in the systems we build, when they run.

The introduction of the MOD language suggests a stepwise design methodology. In early stages in the design process, MODs will most often be used by domain experts and analysts to describe possible snapshots of a system. In doing so, they would stipulate that the system should at least be able to exhibit the examples shown in the MODs. That is, only positive example MODs will be used in the early stages of the design process. As the process matures, knowledge will become available about structures that should not be possible, so the initial set of positive example MODs could be refined with negative examples. Finally, in later stages, analysts will be confident enough to define positive and negative invariant MODs.

The MOD language and this design process are inspired by an analogous design process for behavioral specifications. There, domain experts may provide positive example execution traces, which the system should allow, negative example traces, which the system should not allow, invariant traces, which all system executions should include, and negative invariant traces, which no execution that allows them can be extended to an accepted one. Then, software engineers are responsible for designing a state-machine that will satisfy these multi-modal trace requirements, and model-checking techniques can be used to verify them. Such concrete multi-modal traces can be specified, e.g., using live sequence charts (LSC) [5,12] (see the related work section).

To conclude, we believe that the MOD language can be used not only by software engineers but also by domain experts and analysts, in particular during early requirements phases of object-oriented systems. Moreover, the language supports a stepwise design methodology and can serve as a rich and formal means of communication between the domain experts and the software engineers responsible for the system’s design. The verification technique we provide would aid the engineers in checking that their design indeed meets the concrete requirements set by the MODs defined by the domain experts.

7.5 Synthesis and Unsatisfiable Cores

The most important future research we consider relates to a synthesis problem: given an MOD specification MS , find cd such that $cd \models MS$, if any. That is, we aim to develop an algorithm that takes as input a set of multi-modal object diagrams, made of positive and negative examples as well as positive and negative invariants, and outputs a CD that satisfies it (or reports that such a CD does not exist!). Note that for a classical set of ODs without modalities, each specifying a positive example, this problem is trivial, but for a multi-modal set it is much harder (and interesting). Also note that in many cases, there will be many possible solutions to the synthesis problem; we may be interested in synthesizing a satisfying CD that is minimal with regard to some cost function (e.g., depth or breadth of inheritance tree).

In the case where a satisfying CD cannot be synthesized, we will be interested in the related problem of finding an unsatisfiable core: a minimal subset of the MOD specification that has no satisfying CD (note that there may be more than one unsatisfiable core). The computation of an unsatisfiable core is a well known problem for SAT solvers. Unsatisfiable cores are essential means for the debugging of MOD specifications.

We hope to present the results of this research on synthesis and unsatisfiable cores for MOD specifications in a future paper.

8 Related Work

We discuss related work in adding modalities to existing modeling languages, in specifying constraints on ODs and combining them and OCL, in using Alloy for the analysis of class diagrams, and in other analysis problems related to class diagrams.

The idea that system models should include not only positive examples but also negative examples and positive and negative invariants is not new. This idea has been presented and investigated before in the context of behavioral models, in particular in scenario-based specifications. For example, the language of live sequence charts (LSC) [512] extends classical message sequence charts (MSC) with universal and existential modalities, allowing to specify scenarios that must happen, scenarios that may happen, and scenarios that should never happen. In other variants of MSC [34], negative scenarios are used as a means for requirements elicitation and refinement. As in the case of MODs, the addition of modalities to the modeling language at hand, in this case, message sequence charts, results in a more expressive and useful language. It also comes with a price, in the form of a computationally expensive analysis (see, e.g., synthesis from LSC [1113]). Scenario-based specifications notwithstanding, we are not aware of any other study that investigates the addition of modalities in the context of structural system models, as we do with the introduction of modal object diagrams in the present paper.

Constraint diagrams [17] are a visual notation for specifying invariant constraints on object-oriented models, which can be viewed as a generalization of

instance (object) diagrams, partly inspired by Venn diagrams. One may consider constraint diagrams to be similar to MODs, as both express constraints on object-oriented models. However, the two languages are fundamentally different. Constraint diagrams have their own visual notation while MODs only extend existing visual or textual notation with stereotypes. The invariants of constraint diagrams can be compared to the invariants of OCL (see Sect. 7.3 for a discussion on the relationship between OCL and MOD). To the best of our knowledge, based on [17], constraint diagrams cannot specify examples and have no explicit support for negation.

An integration of object diagrams and OCL for the specification of object-oriented systems is part of the definition of UML/P (see [27] ch. 5.3). This work proposes the embedding of object diagrams into OCL to, e.g., define the context of invariants, describe pre- and post-conditions, or specify relations between object diagrams (e.g., implication). Furthermore, elements like attributes and association links in object diagrams can be accessed from within the OCL/P syntax, relating the ODs to a system state. The semantics of logical conjunctives like `&&`, `||`, `implies`, etc. between ODs are informally given in [27]. Thus, the modalities of MOD can be expressed as an OCL/P predicate referencing UML/P ODs. Moreover, the other direction, of embedding OCL expressions into object diagrams, is defined too. For example, the language permits the definition of OCL/P variables inside object diagrams, e.g., to enable parametrized ODs, similar to the extension mentioned in Sect. 6.2. Our work is to a great extent inspired by these ideas. The UML/P language of [27] is far more expressive than MOD, however, it has no supporting reasoning mechanism and implementation. MOD can be viewed as a variant of UML/P ODs and their combination with OCL.

Some previous works consider the use of Alloy for the analysis of CDs (see, e.g., [130]). These works focus on the formal definition of the transformation of a single CD to an Alloy module at the level of a meta-model and on its implementation using a transformation language. Possible applications of the use of Alloy to analyze a given CD are not discussed in depth in these works. In contrast, the input for our transformation consists not only of a class diagram but also of an object diagram (or a set of object diagrams). Moreover, the transformation itself is different, as it follows a pragmatic approach: we are not suggesting a meta-model level framework for general transformations but instead focus on solving the concrete verification problem we have at hand. Defining and implementing our transformation using QVT or other transformation language such as ATL [16] is possible, but is outside the focus of our work.

A different use of Alloy is considered in [29], where the authors present a meta-model directed model completion feature, in the context of code completion support in editors of domain-specific modeling languages. Although the setup and motivation are very different than ours, this work is somewhat similar to our work, specifically in the way predicates are used to define partial models.

Some previous works consider various analysis problems related to CDs (see, e.g., [210,2131,33]). These include the finite satisfiability problem, the consistency of UML models (with or without OCL constraints), the identification

of implicit consequences etc. Some of these use a direct translation into SAT and provide experimental performance results [31]. Others use Description Logic (DL) as their underlying formalism [33]. Some works include no implementation but present theoretical results about the decidability and complexity of the problems at hand. In contrast, we introduce a modal extension to the OD language and consider the problem of verifying that a given CD models a multi-modal specification. We provide a solution, in a bounded scope, using a reduction to an Alloy module and its analysis with a SAT solver.

Finally, in another paper in this conference [20] we have defined CDDiff, a semantic differencing operator for CDs (used for semantic model comparison in the context of model evolution), and have implemented it using a translation to Alloy. However, the translation we use for CDDiff is very different than the one we use here. The input for CDDiff consists of two CDs and its output is an OD that represents an OM that is in the semantics of the first CD and not in the semantics of the second. The input for MOD verification is a CD and an OD. While in CDDiff, each of the two input CDs is represented using a predicate, here we use the input CD as a base and the input OD induces a predicate that constrains it.

9 Conclusion

We introduced modal object diagrams, as an expressive extension to the classical object diagrams language. Moreover, we have presented a verification technique that can be used to verify, in a bounded scope, whether a given class diagram satisfies a multi-modal object diagram specification. We discussed a stepwise design process, where domain experts and analysts provide MODs while software engineers are responsible for designing class diagrams that satisfy them. The extended language and the verification technique are fully implemented in a prototype Eclipse plug-in.

The tradeoff of formality and expressiveness vs. intuitiveness and ease of use is a major challenge in modeling languages design. We believe that MOD addresses this tradeoff well: it is expressive enough to be valuable in specifying structural requirements of object-oriented systems, yet it is also intuitive and simple enough to be attractive to engineers.

Finally, we considered the advantages and limitations of our work. As discussed in Sect. 7, future work includes the development of heuristics to improve the performance of our verification technique and allow it to scale, the embedding of a subset of OCL inside the MOD language in order to extend its expressive power, performing case studies that will evaluate the use of MOD in the design of real-world systems, and the investigation of the problem of synthesizing a CD from a multi-modal object diagram specification.

Acknowledgments. We are grateful to Martin Schindler for defining the MontiCore language support for ODs and CDs. We thank Smadar Szekely and Guy Weiss for their expert advice on Eclipse plug-in development. We thank Mira Balaban, David Lo, and the anonymous reviewers for comments on a draft of this paper.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. *Software and Systems Modeling* 9(1), 69–86 (2010)
2. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. *Artif. Intell.* 168(1-2), 70–118 (2005)
3. Broy, M., Cengarle, M.V., Grönniger, H., Rumpe, B.: Definition of the System Model. In: Lano, K. (ed.) *UML 2 Semantics and Applications*. Wiley, Chichester (2009)
4. Cengarle, M.V., Grönniger, H., Rumpe, B.: System Model Semantics of Class Diagrams. *Informatik-Bericht 2008-05*, Technische Universität Braunschweig (2008)
5. Damm, W., Harel, D.: LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design* 19(1), 45–80 (2001)
6. Dresden OCL, <http://www.reuseware.org/index.php/DresdenOCL> (accessed April 2011)
7. Eclipse UML2 project, <http://www.eclipse.org/modeling/mdt/?project=uml2> (accessed April 2011)
8. Evans, A., France, R.B., Peng, S.-L.: The UML as a Formal Modeling Notation. In: Bézivin, J., Muller, P.-A. (eds.) *UML 1998*. LNCS, vol. 1618, pp. 336–348. Springer, Heidelberg (1999)
9. FreeMarker, <http://freemarker.org/> (accessed April 2011)
10. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, independence and consequences in UML and OCL models. In: Dubois, C. (ed.) *TAP 2009*. LNCS, vol. 5668, pp. 90–104. Springer, Heidelberg (2009)
11. Harel, D., Kugler, H.: Synthesizing state-based object systems from LSC specifications. *Int. J. Found. Comput. Sci.* 13(1), 5–51 (2002)
12. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)* 7(2), 237–252 (2008)
13. Harel, D., Maoz, S., Segall, I.: Some Results on the Expressive Power and Complexity of LSCs. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) *Pillars of Computer Science*. LNCS, vol. 4800, pp. 351–366. Springer, Heidelberg (2008)
14. IBM Rational Software Architect (RSA), <http://www.ibm.com/developerworks/rational/products/rsa/> (accessed April 2011)
15. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2006)
16. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. *Sci. Comput. Program.* 72(1-2), 31–39 (2008)
17. Kent, S.: Constraint diagrams: Visualizing assertions in object-oriented models. In: *OOPSLA*, pp. 327–341 (1997)
18. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer (STTT)* 12(5), 353–372 (2010)
19. Kuss, E.: Using Alloy Analyzer for automated consistency checks between UML/P class and object diagrams. Master's thesis, Software Engineering, RWTH Aachen, Germany (2010) (in German)
20. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: Semantic differencing for class diagrams. In: Mezini, M. (ed.) *ECOOP 2011*. LNCS, vol. 6813, pp. 230–254. Springer, Heidelberg (2011)

21. Maraee, A., Balaban, M.: Efficient reasoning about finite satisfiability of UML class diagrams with constrained generalization sets. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 17–31. Springer, Heidelberg (2007)
22. MOD project materials, <http://www.se-rwth.de/materials/mod/>
23. MontiCore project, <http://www.monticore.org/>
24. Nierstrasz, O.: Ten things I hate about object-oriented programming. ECOOP 2010 9(5) (September 2010) (editorial [Banquet speech given at ECOOP 2010, Maribor, June 24 2010])
25. OMG (Object Management Group). Object Constraint Language (OCL), <http://www.omg.org/spec/OCL/2.2/> (accessed May 2011)
26. Poseidon for UML, <http://www.gentleware.com/> (accessed May 2011)
27. Rumpe, B.: Modellierung mit UML. Springer, Heidelberg (2004)
28. SAT4J project, <http://www.sat4j.org/> (accessed May 2011)
29. Sen, S., Baudry, B., Vangheluwe, H.: Towards domain-specific model editors with automatic model completion. *Simulation* 86(2), 109–126 (2010)
30. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to alloy and back again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010)
31. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: DATE, pp. 1341–1344. IEEE, Los Alamitos (2010)
32. Sparx Systems Enterprise Architect, <http://www.sparxsystems.com/> (accessed May 2011)
33. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
34. Uchitel, S., Kramer, J., Magee, J.: Negative scenarios for implied scenario elicitation. In: SIGSOFT FSE, pp. 109–118. ACM, New York (2002)

Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks^{*}

Robert L. Bocchino Jr.¹ and Vikram S. Adve²

¹ Carnegie Mellon University

² University of Illinois at Urbana-Champaign

Abstract. Object-oriented frameworks can make parallel programming easier by providing generic parallel algorithms such as map, reduce, or pipeline and letting the user fill in the details with sequential code. However, such frameworks can produce incorrect behavior if they are not carefully used, e.g., if a user-supplied function performs an unsynchronized access to a global variable. We develop novel techniques that can prevent such errors. Building on a language (Deterministic Parallel Java, or DPJ) with an expressive region-based type and effect system, we show how to write a framework API that enables sound reasoning about the effects of unknown user-supplied methods. We also describe novel extensions to DPJ that enable generic types and effects while retaining soundness. We present a formal semantics and soundness properties for the language. Finally, we describe an evaluation showing that our technique can express three parallel frameworks and three realistic parallel algorithms using those frameworks.

1 Introduction

The emergence of commodity multicore systems is driving parallel programming into the mainstream, posing new productivity, correctness, and performance challenges for programmers who are used to writing sequential code. One way to alleviate these challenges is to use object-oriented frameworks. The framework writer provides most of the code for parallel construction and manipulation of generic data structures; for generic parallel algorithms such as map, reduce, or scan; or for generic parallel coordination patterns such as pipelines. The user fills in the missing pieces with code that is applied in parallel by the framework. Examples include the algorithm templates in Intel's Threading Building Blocks (TBB) [28] and Java's ParallelArray [1]. Such frameworks are usually easier to reason about than general parallel programming because the user only has to write sequential code, letting the framework orchestrate the parallelism.

^{*} This work was supported by the National Science Foundation under grants CCF 07-02724 and CNS 07-20772, and by Intel, Microsoft, and the University of Illinois through UPCRC Illinois. Robert Bocchino is supported by a Computing Innovation Fellowship.

However, state-of-the-art frameworks give no guarantee of noninterference of effect, and this a serious deficiency in terms of correctness and program understanding. For example, `ParallelArray`'s `apply` method applies an arbitrary user-specified function to each element of the array. If that operation performs an unsynchronized update to a global, then a race will result. It would be much better if (1) the framework developer could write an API expressing a contract (for example, the function provided to `apply` has no potentially interfering effects on shared state); and (2) the compiler could check that the contract is met by all code supplied by the user to the framework. While several tools and techniques exist that support writing and checking assertions at interface boundaries [20, 25, 35], these ideas have not yet been applied to enforce *parallel noninterference*. Doing so poses several challenges:

1. *Maintaining disjointness.* Useful parallel frameworks need to support parallel updates on contained objects. For example, we would like a `ParallelArray` of distinct objects, where the user can define a method that updates an element, and ask the framework to apply it to each distinct object in parallel. To do this safely, the framework must ensure that the objects are really distinct; otherwise the same object could be updated in two parallel iterations, causing a race. For a language like Java with reference aliasing, disjointness of reference is a nontrivial property.
2. *Constraining the effects of user-supplied methods.* For a parallel update traversal over the objects in a framework, disjointness of reference is necessary but not sufficient to ensure noninterference. The framework must also ensure that the effects of the user-supplied methods do not interfere, for example by updating a global variable, or by following a link from one contained object to another.
3. *Making the types and effects generic.* Because different uses of the framework need user-supplied methods with different effects, the framework should constrain the effects of user-supplied methods as little as possible while retaining soundness. For example, one use of `apply` may write into each object only, while another may read shared data and write into each object. The framework should also be generic in the type of the contained objects. These requirements pose challenges when the framework author needs information about the type of the contained objects and the effect of user-supplied methods in order to provide a noninterference guarantee.
4. *Writing the framework implementation.* The framework author must ensure that the internal framework implementation guarantees safe parallelism, given that the API is enforced. For example, the framework implementation must ensure that any parallel loop inside the framework iterates exactly once over each contained object.

Notice that the first three challenges are about defining a framework *API* that enables sound reasoning about uses of the framework, while the fourth challenge is about writing a framework *implementation*.

In this work we primarily address the first three challenges, i.e., we show how to write a framework API so that the framework author can reason soundly

about interference of effect in arbitrary instantiations of the framework, with unknown user-supplied methods and generic type bindings. We build on Deterministic Parallel Java (DPJ) [6,7], which expresses effects in terms of *regions* that partition the heap. Regions provide an intuitive and flexible way to express and check effects.

As to the fourth challenge, we state the properties (type preservation, effect preservation, and noninterference) that a correct framework must satisfy. In many cases DPJ can verify those properties. In some cases, however, the DPJ effect system may be insufficiently expressive to guarantee disjointness of effect. Here the framework author is free to use a different strategy, such as program logic [14,15], testing, or model checking, to verify disjointness. Such checking is completely hidden from the user of the framework, so that the user gets a strong guarantee: if the program type checks, then there is no interference.

Our contributions are the following:

1. We show how to write a framework API using DPJ as described in [6] so that the framework implementer has all the information necessary to guarantee disjointness of reference and sound effects for user-supplied methods.
2. We show how to extend DPJ to add generic effects and generic types, making the frameworks more general and useful. For the effects, we add *effect variables*, together with *effect constraints* to enforce disjointness of effect. For generic types, we introduce *type region parameters*, a form of type constructor, to guarantee disjointness and soundness of effect, without knowing the exact type bound to type variables.
3. We sketch the formal semantics of a core subset of the extended language and formally state the soundness results. The full semantics and proofs are stated in the first author's Ph.D. thesis [5].
4. We state the requirements for a correct framework implementation, such that if these requirements hold, then noninterference is guaranteed for the entire program. We also show how to use a combination of the DPJ type system and external reasoning to check the requirements informally. We leave as future work the formal verification of the requirements.

To evaluate our techniques, we used them to write three parallel frameworks (Parallel Array, Parallel Tree, and Pipeline) and three applications using those frameworks. We found that the techniques are expressive enough to capture realistic parallel algorithms. We also found that the extra annotations required by the system are fairly simple for framework users and, while more complicated for framework writers, are not unduly burdensome.

2 Background

DPJ. We begin with a brief introduction to DPJ [6]. DPJ uses *regions* to specify access to the heap: every class field and array cell lies in a single region, and distinct regions represent disjoint collections of memory locations. A region can be a declared name r , or a colon-separated list of names, such as $r_1 : r_2 : r_3$, called

```

1 public class Node<region R> {
2     int data in R;
3     Node<*> next in R;
4     public Node(int data, Node<R> next) pure { this.data = data; this.next = next; }
5 }

```

Fig. 1. Node class that will serve as a running example

a *region path list* (RPL). RPLs give rise to a natural *nesting* structure: one RPL is “under” another if the second is a prefix of the first. For example, $r_1:r_2$ is under r_1 . Nesting is useful for expressing *effects* (i.e., what regions are read and written by a particular program statement). The set of all regions under an RPL R is denoted $R:*$.

Figure 1 defines a simple list node class that we will also use in subsequent sections. The class has one region parameter R . Fields `data` and `next` are both placed in region R . When class `Node` is instantiated into a type, both fields will be in the region given as the argument to R in the type. The effect of the constructor is declared `pure` (no effect), because in DPJ an object is not visible to the rest of the program until the constructor returns, so constructors do not have to report their effects on the constructed object. In general, a method must summarize its effects; if there is no effect summary, the default is “writes the whole heap.”

Figure 2 shows a simple container class, `NodePair`, that stores a pair of list nodes. Line 2 declares region names `First` and `Second`. Lines 3–4 instantiate `Node` types using these names. Line 12 reads field `first`, located in region `First` (line 3). It also writes the `data` field of `first`, which is located in region R (line 2 of Figure 1) after substituting `First` for R , from the type of `first` (line 3 of Figure 2). Thus the effect of the write is `writes First`. Writes cover reads in DPJ, so the whole effect of line 12 may be summarized as `writes First`, as shown. The same reasoning gives the effect `writes Second` shown in line 13. Because `First` and `Second` are distinct names, the compiler can conclude that the updates in lines 12 and 13 are disjoint. With these features, together with additional features for arrays, divide and conquer parallelism, and commutative operations, DPJ can express important patterns of parallelism [6].

```

1 class NodePair {
2     region First, Second;
3     Node<First> first in First;
4     Node<Second> second in Second;
5     NodePair(Node<First> first,
6             Node<Second> second) pure {
7         this.first = first;
8         this.second = second;
9     }
10    void updateNodes(int fd, int sd) {
11        cobegin {
12            first.data = fd; // writes First
13            second.data = sd; // writes Second
14        }
15    }
16 }

```

Fig. 2. Using region parameters to distinguish object instances

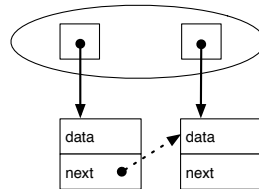


Fig. 3. A potential race caused by cross links. A race can occur if a task operating on the left-hand reference follows the dashed arrow to access the right-hand object.

Difficulties with Region-Based Systems. Region-based effect systems can be quite expressive, and they are a natural choice for writing safe object-oriented frameworks. However, existing systems impose significant limitations. As we will see, by shifting some of the burden of guaranteeing noninterference from the type system to the framework, we can overcome some of these limitations.

One limitation is that to guarantee soundness we have to prohibit swapping of `first` and `second` in the example:

```
void swap() {
    Node<First> tmp = first;
    first = second;      // Illegal: Can't assign Node<Second> to Node<First>
    second = tmp;       // Illegal: Can't assign Node<First> to Node<Second>
}
```

If we could do such an assignment, then we could have multiple references with conflicting types pointing to the same data, and we would no longer be able to draw sound conclusions about effects.

For this reason, DPJ and other region-based systems [23, 9] use wildcard types. For example, in lines 3–4 of Figure 2, we could have written both types `Node<*>`, where `*` stands in for any region. Now the swapping shown above is sound. However, we have lost the ability to distinguish writes to `first.data` and `second.data` using the type system, because now all we know is that the writes are to `*`. So in this case, the state of the art in region-based type systems forces us to choose: either we can prove that two references don't alias, or we can swap the two references, but not both. Notice, however, that (1) regions `First` and `Second` are distinct coming into the constructor (line 5); and (2) the `swap` operation preserves the distinctness of `First` and `Second` in the dynamic types of `first` and `second`. So in fact it is possible both to allow the swap and to prove disjointness, although the type system can't do both.

In fact, the situation is worse than this. As shown in Figure 3, a `NodePair` holding distinct list nodes can have cross links. The effect system must ensure that when following the references to access the objects in parallel, cross links are never followed to update the same object. Further, we probably don't want to encode the write to `data` into the framework implementation, as shown in lines 12–13. Instead, as discussed in the introduction, we would like to express the operation abstractly and let the user supply the specific operation. We must constrain the effects of the user-supplied method so that for any user-supplied method, this kind of interference cannot happen. Finally, we don't really want a `NodePair` class; instead, we want a `Pair<T>`, where `T` is a generic type.

3 Safe, Reusable Parallel Frameworks

We now show how to address the challenges discussed above to write safe, reusable parallel frameworks. First we show how to write a container API that supports sound reasoning about effects for a container specialized to list nodes. Second, we show how to extend the effect system to make the container API

generic. Third, we address the problem of writing a correct framework implementation. Although most of this section focuses on the container example, the work is not specific to containers. In the next section, we formalize the techniques in general terms, without specifically considering disjoint containers. In Section 5 we use the techniques to write a framework for pipeline parallelism.

3.1 A List Node Container

In this section, we show how to use the DPJ effect system as previously described [6] to write a container API that stores `Node` objects and allows safe parallel updates to the stored objects. The API generalizes the trivial `NodePair` class from the previous section into an arbitrary container. The container *implementation* is not specified; it could be any container (set, list, tree, etc.). The point is that we will be able to write an API for a container that (1) holds list nodes, which may have cross links between them, as shown in Figure 3; and (2) allows update operations on the nodes to be done *safely in parallel*, despite the presence of the cross links. We will extend the example further to a more generic (and more useful) container in later sections. Writing the list node container API presents two problems: maintaining disjointness and reasoning about effects.

Maintaining Disjointness. To enable parallel update traversals over the contained nodes, we wish the container to have the following two properties. First, at runtime, every node element e_i stored in the container either is `null` or points to an object with a region R_i in its type. Second, for any $i \neq j$, if e_i and e_j are both non-`null`, then R_i and R_j are *disjoint* (i.e., R_i and R_j refer to nonintersecting sets of regions). In general, we call a container “disjoint” if it satisfies both properties for its elements e_i . Note that the `NodePair` container from Section 2 satisfies this definition. Disjointness ensures that parallel tasks that update the regions of different elements are noninterfering.

To enforce disjointness, we use the following strategy: (1) every container starts empty and so is trivially disjoint; and (2) every operation provided by the disjoint container API is disjointness-preserving (takes a disjoint container to another disjoint container). By a simple induction, we can then conclude that the container is disjoint throughout its lifetime. The hard part is guaranteeing property (2). In some cases, this problem may be reduced entirely to the problem of writing a correct framework implementation (Section 3.3). Examples include

```

1 public interface NodeContainer<region RN,RC | RN:* # RC> {
2     /* One linear container from another */
3     public NodeContainer(NodeContainer<RN,RC> c) writes RC;
4
5     /* Controlled creation of contents */
6     public NodeContainer(NodeFactory fact, int size) writes RC;
7     public interface NodeFactory { public <region R>Node<R> create(int i) pure; }
8
9     /* Parallel operation on all elements */
10    public void performOnAll(Operation op) reads RC writes RN:*;
11    public interface Operation { public <region R>void operateOn(Node<R> elt) writes R; }
12 }

```

Fig. 4. Framework API for a disjoint list node container

```

1  /* Implement factory interface */
2  public class MyFactory implements NodeContainer.NodeFactory {
3      public <region R>Node<R> create(int i) { return new Node<R>(i, null); }
4  }
5  /* Declare region names A and B */
6  region A, B;
7  /* Use factory to make NodeContainer<A,B> with 10 elements */
8  NodeContainer<A,B> c = new NodeArray<A,B>(new MyFactory(), 10);

```

Fig. 5. Use of the NodeFactory API from Figure 4

tree rebalancing and array shuffling operations that modify only the internal structure of the container. In other cases, the framework implementation may need to cooperate with user-provided code. An example is putting things into a container: the user must have some control over objects placed in the container, but the framework must support sound reasoning about disjointness.

We have explored the following two strategies for controlling disjointness: building one disjoint container from another and controlled creation of contained objects. Figure 4 shows an API for a list node container that illustrates these strategies. There are two region parameters, RN for the nodes of the container and RC for the container itself. In line 1, we use a *region parameter constraint* [6,9] to require that for any instantiation of `NodeContainer` that binds R_1 to RN and R_2 to RC, $R_1 : *$ and R_2 are disjoint. This ensures that reading the container to traverse the elements does not interfere with updating the contained objects.

Building one disjoint container from another. Line 3 of Figure 4 illustrates this strategy: it says that given an object of type `NodeContainer<RN,RC>` we can create another one. An example is creating a tree from an array or set. An important special case in DPJ is creating a disjoint container from an *index-parameterized array* [6], which supports parallel update traversals but does not support reshuffling of elements. (This is exactly the problem discussed in Section 2, just with array cells rather than fields.) A disjoint container created from an index-parameterized array supports disjointness-preserving operations, including shuffling, by doing them *internally* within the framework.

Controlled creation of contained objects. Lines 6–7 of Figure 4 illustrate this strategy, for an interface to `NodeContainer` that could be implemented in different ways (array, tree, etc). The container implementation does the actual object creation, but the user specifies the number of objects to create and provides a factory method that creates the i th object. For example, a use could look as shown in Figure 5, assuming a class `NodeArray` that implements `NodeContainer`.

The important thing here is that the factory method must really create a new object and not, for example, just fetch some object reference from the heap and store it into the container over and over. The framework author can enforce this requirement by judicious use of a *method region parameter*.¹ In line 7 of Figure 4, the return type of the factory method is written in terms of a parameter R that is in scope only in that method. Further, no reference assignable to type `Node<R>`

¹ Method region parameters are not claimed as new in this work; the contribution here is to use them for safe factory methods.

enters the method. Therefore, the only way a `Node<R>` can escape the method is if it is created inside the method. The effect of this strategy is to hide the actual regions R_i in the types of the created objects from the user code: to create a new object in R_i , the framework binds R_i to R and calls the user’s factory method. The factory method doesn’t know what R_i is, except that it is bound to R . On the other hand, the framework doesn’t know what the factory method is, except that when called with $R = R_i$ it creates a new `Node<R_i>`.

Reasoning about Effects. Lines 10–11 of Figure 4 show the part of the API that allows the user to define a method and then pass that method into the container to be applied in parallel to all contained objects. For example, given reference `c` of type `NodeContainer<A,B>`, the user could do this:

```
public class MyOperation implements NodeContainer.Operation {
    public <region R>void operateOn(Node<R> elt) writes R { ++elt.data; }
}
c.performOnAll(new MyOperation());
```

This code increments the `data` field of each of the objects stored in `c` in parallel.

Effect of `operateOn`. In the definition of the abstract `operateOn` method in the `Operation` interface (line 11 of Figure 4), we again use a method region parameter R . We write the type of the formal parameter `elt` as `Node<R>`, and we specify the effect as `writes R`. This causes two things to happen. First, the DPJ type system requires that any user-supplied method implementing `operateOn` must have a declared effect that is a *subeffect* of `writes R`. That means all the effects represented by E_2 are also represented by E_1 . For example, `reads R` is allowed, but reading or writing some other region is not.² In particular, if `MyOperation` had contained the statement `++elt.next.data`, the effect would be `writes *`, which is not a subeffect of `writes R`, and the compiler would catch the error. Thus, the effect annotations prohibit using cross links to cause a race, as shown in Figure 3. Second, the API techniques discussed above ensure that the regions in the contained elements are disjoint. Together, these two facts guarantee that the effects of different parallel tasks operating on different elements are noninterfering.

Effect of `performOnAll`. In Figure 4, we have written the effect of `performOnAll` as `reads RC writes RN:*`. This is correct if, for a particular implementation of the interface, (1) each element i has type `Node<R_i>`, where R_i is under RN ; and (2) the implementation of `performOnAll` reads the container and applies the user’s `operateOn` method to the elements. As discussed further in Section 3.3, the framework writer is responsible for ensuring that both facts are true. Further, if the framework internals are written in DPJ, then DPJ can verify these facts.

3.2 Getting More Flexibility

We now show how to generalize the list node container to a generic container. This requires some extensions to the DPJ effect system.

Generic Effects. In the previous API, the effects of `operateOn` are overly restricted. For instance, what if the user wants `operateOn` to read some region

² The relevant rules for subeffects are given formally in the next section.

```

1 public interface Operation<effect E> {
2     public <region R>void operateOn(Node<R> elt) writes R effect E;
3 }
4
5 public <effect E | effect E # reads RC writes RN:* effect E>
6     void performOnAll(Operation<effect E> op) reads RC writes RN:* effect E;

```

Fig. 6. Making the effects of the `Operation` interface generic

```

1 public class MyOperation implements NodeContainer.Operation<reads Global> {
2     public <region R>void operateOn(Node<R> elt) reads Global writes R {
3         elt.data = global; // global is in region Global
4     }
5 }
6 c.<reads Global>performOnAll(new MyOperation());

```

Fig. 7. Use of the API from Figure 6

R' disjoint from $R:*$, where R is the region bound to RN in the instantiation of the framework interface? That is safe and should be allowed. Yet it is disallowed by the effect specification `writes R` in the API.

To address this problem, we use effect polymorphism [24]. As shown in Figure 6, we give the `Operation` interface an *effect parameter* E that becomes bound to an actual effect (for example, `Operation<reads r>`) when the interface is instantiated into a type. To make this strategy work, we need to solve two problems: (1) constraining the effect arguments to ensure noninterference; and (2) ensuring soundness of subtyping.

Constraining the effect arguments. The framework cannot let the effect variable E become bound to an arbitrary effect in the user's code, because that would re-introduce a user-supplied method with unregulated effects. Instead, we use an *effect constraint* that restricts the effect of the user-supplied method, as shown in Figure 6. We give the `Operation` interface (line 1) an effect variable E . We also give the `performOnAll` method (lines 5–6) a *constrained method effect parameter* E . After the parameter declaration is a constraint specifying that the effect bound to E must be noninterfering with `reads RC writes RN:* effect E`. This constraint ensures that the supplied effect will not interfere with any of (1) the effect `reads RC` of reading fields of the container; (2) the effect `writes RN:*` of updating the nodes; or (3) itself. The last constraint implies that either E is a read-only effect, or it is an update operation such as an atomic set insert that commutes with itself [6]. As an example, Figure 7 shows a user-supplied method that puts all the `Node` objects in region `A` and reads region `Global` to initialize all the objects with the same global value.

Soundness of subtyping. Once we add class effect parameters, we need a rule for deciding if $C\langle E_1 \rangle$ is a subtype of $C\langle E_2 \rangle$, where E_1 and E_2 are effects. We could require that E_1 and E_2 be identical effects. However, this is more restrictive than necessary. While that alone might be acceptable, it turns out this approach is also not sound if done in the obvious way, as discussed in Section 4.5. Instead, we let E_1 be a subeffect of E_2 . For example, E_1 could be `writes r_1` and E_2 could be `writes r_1, r_2` .

This approach introduces a subtle requirement for preserving consistency of types. For example, consider the following snippet:

```
class C<effect E> { C<effect E> f; }
C<writes r> x = new C<pure>();
```

By the subtyping rule stated above, this code is legal. But then what is the static type of `x.f`? The obvious answer is `C<writes r>` (substituting `writes r` from the type of `x` for `E` in the declaration of `f`), but this is incorrect. For in that case, a reference of type `C<writes r>` could be legally assigned to `x.f`. But the dynamic type of `x.f` is `C<pure>`, and `writes r` is not a subeffect of `pure`, so the assignment violates type preservation.

The solution we adopt to make the static type of `x.f` `C<effect E'>`, where `E'` is a fresh effect parameter (called a *capture parameter*). The tricky thing here is that *all nonempty effects must be captured when substituted for an effect parameter in a type*. This is because all nonempty effects are essentially wildcards: the runtime effect could be equal to the static effect, or it could be empty (or possibly something else, e.g., `reads R` instead of `writes R`, or `reads R1` instead of `reads R1, R2`). Our solution follows in the same vein as generic wildcards [10] (which stand in for several types) and DPJ's partially-specified RPLs [5] (which stand in for several RPLs).

Generic Types. It is also too restrictive to make the class specialized to list nodes. Instead, we want a class `DisjointContainer<type T, region RC>` with a generic type `T`. Notice, however, that the region argument to the `Node` type is essential to writing the API. For example, in writing the `NodeFactory` interface of Figure 4, we used a method-local parameter `R` in the return type of `create`. If we just replaced that type with an ordinary type variable `T`, then we would not be able to write the node factory pattern at all. A similar issue occurs in writing the effect of `performOnAll`.

To solve this problem, we use a type constructor [3, 26] that takes a region argument. A type variable can be declared `T<region R>`, where `R` declares a fresh parameter. We call `R` a *type region parameter*. When a type `T` becomes bound to a type variable `T`, `T` must have at least one region argument, and `R` represents the first region argument. We write uses of the variable `T` as `T<r>`, where `r` is a valid region in scope. `T<r>` represents the same type with the region in its first argument position replaced by `r`. Notice that according to this rule the parameter `R` is a valid region, and `T<R>` represents the unmodified type provided as the argument to the variable. For convenience, a bare use of `T` is allowed, and this is equivalent to `T<R>`. Our language also supports multiple type region parameters for a variable `T`; this straightforward extension is discussed in [5].

Final Container API. Figure 8 shows the final disjoint container API. Line 1 declares an interface `DisjointContainer` with one type parameter `T` and one region parameter `Cont`. The type parameter has one region parameter `Elt` that names the first region argument of the type bound to `T`. In line 8, we write `T<R>` to require that the return type of `create` have the method region parameter `R` as its first region argument. In line 12, the region `Elt` is available to write the


```

1 public interface DisjointContainer<type T<region Elt>, region RC | Elt:* # RC> {
2
3     public DisjointContainer(DisjointContainer<T,RC> cont) writes RC;
4
5     public <effect E # writes RC effect E>
6         DisjointContainer(Factory<T, effect E> fact, int size) writes RC effect E;
7     public interface Factory<type T<region Elt>, effect E> {
8         public <region R>T<R> create(int i) effect E;
9     }
10
11     public <effect E # reads RC writes Elt:* effect E>
12         void performOnAll(Operation<T,effect E> op) reads RC writes Elt:* effect E;
13     public interface Operation<type T<region Elt>, effect E> {
14         public <region R>void operateOn(T<R> elt) writes R effect E;
15     }
16 }

```

Fig. 8. API for a disjoint container with generic types and effects

```

1 public class MyOperation implements
2     DisjointContainer.Operation<Node<A>,pure> {
3     public <region R>void operateOn(Node<R> elt) writes R { ++elt.data; }
4 }
5 c.performOnAll(new MyOperation());

```

Fig. 9. Use of the API from Figure 8

effects of `performOnAll`. We do the same thing for the type parameter of the `Operation` interface, in line 13.

Figure 9 shows an example implementation of `operateOn`, assuming `c` has type `DisjointContainer<Node<A>,B>`. The effect argument in line 2 is `pure`, because no effect is needed for this implementation of `operateOn`, except for `writes R`, which is already given by the interface (line 14 of Figure 8). The effect of the call to `performOnAll` in line 5 is `reads B writes A:*`.

3.3 Writing the Framework Implementation

We now address the problem of writing a correct framework implementation. The framework must ensure three properties: type preservation, effect preservation, and noninterference of effect. The key point is that *the API design discussed in the previous sections provides all the information needed to reason soundly about these three properties, even in the presence of unknown user-supplied methods*. Further, the framework author can write the framework in DPJ, thereby using DPJ to check some or all of these properties. However, so long as the properties hold for all user-visible types and effects, the framework author may use *internal* operations, such as swapping references with disjoint regions, that DPJ alone cannot prove correct.

Type Preservation. Type preservation means that the static types of variables agree with the dynamic types of the references they store. If the framework is written in DPJ, then this property will be checked “for free,” unless the framework does an assignment (using a cast) that violates the typing rules. Such type casts produce a warning, but the code compiles and runs.

The DPJ subtyping rules are flexible, so we anticipate that unsound assignments will rarely be needed in practice. A more likely case is that casts are used to interface with non-DPJ code. For example, pre-Java 5 code implementing a container might represent the container elements as references to `Object` and require that they be cast back to their actual type when removed from the container. For Java code written with generics, such casts should be rare.

Effect Preservation. Effect preservation means that the static effects of statements correctly summarize their dynamic effects. Again, DPJ guarantees this property, so long as (1) type preservation holds; and (2) every method summary covers the effects of the method body. In DPJ, one can always write a correct method summary (`writes *` is always correct), and in fact an incorrect summary causes a compile-time error. So property (2) will hold if property (1) does. If the framework calls into non-DPJ code, then the framework writer must manually ensure that effect preservation holds for the calling code.

Noninterference of Effect. Noninterference means there are no conflicting memory accesses between parallel tasks. While DPJ can establish noninterference in many cases, in some cases it may not be able to, as in the swap example discussed in Section 2. In such cases, the framework author can write code that causes DPJ to produce an interference warning, and use a different technique to show noninterference. As an example, Figure 10 shows an implementation of `DisjointContainer` as a `DPJArrayList`, which is a Java `ArrayList` annotated with region information. In line 4, the type argument to `DPJArrayList` is `Elt:*`, i.e., the type does not specify which cell of the array is in which region, so reshuffling the array is supported. The `performOnAll` method uses the DPJ `foreach` construct (line 8) to iterate in parallel over the elements. We also add a `swap` method, similar to the method discussed in Section 2, for swapping array elements.

```

1 public class DisjointArray<type T<region Elt>, region RC | Elt:* # RC>
2 implements DisjointContainer<T,RC> {
3     /* Internal array representation */
4     private DPJArrayList<T<Elt:*>,RC> elts in RC;
5     /* Implementation of performOnAll */
6     public <effect E | reads RC writes Elt:* effect E>
7         void performOnAll(Operation<T,effect E> op) reads RC writes Elt:* effect E {
8         foreach (int i in 0, elts.size()) { op.operateOn(elts.get(i)); }
9     }
10    /* Swap elements at idx1 and idx2 */
11    public void swap(int idx1, int idx2) writes RC {
12        T<Elt:*> tmp = elts.get(idx1); elts.add(idx1, elts.get(idx2)); elts.add(idx2, tmp);
13    }
14 }

```

Fig. 10. Array implementation of a disjoint container (partial)

To show noninterference, it suffices to establish two things for the `foreach` construct in line 8: (1) for distinct values i , the region in the dynamic type of `elts.get(i)` is distinct; and (2) i attains distinct values i on distinct iterations. The first statement follows from the inductive argument we made in Section 2: to change the shape of the array, we either have to use an inherited creation

method, which preserves disjointness as discussed in Section 3.1, or do a swap, which also preserves disjointness, as can be seen from the implementation in line 11. The second statement follows from the semantics of `foreach` in DPJ.

4 Formal Elements

In this section we formalize the ideas developed in the previous section. We use a sequential core language, which suffices to establish type preservation, effect preservation, and noninterference of effect. As discussed in Section 3.3, a framework designer can use these properties to provide deterministic parallelism or other guarantees for correct framework uses. We give a syntax, static semantics, and dynamic semantics for the core language. Then we state the key soundness results, and sketch the proofs. More detail, including proofs, can be found in [5].

4.1 Syntax

$$\begin{array}{ll}
 \text{Programs } \mathcal{P} & ::= \mathcal{R}^* \mathcal{I}^* \mathcal{C}^* e \\
 \text{Region Names } \mathcal{R} & ::= \text{region } r \\
 \text{Interfaces } \mathcal{I} & ::= \text{interface } I \langle \tau \langle \rho \rangle, \rho, \eta \# E \rangle \{ S^* \} \\
 \text{Classes } \mathcal{C} & ::= \text{class } C \langle \tau \langle \rho \rangle, \rho \rangle \text{ implements } I \langle T, R, E \rangle \{ F^* M^* \} \\
 \text{Method Signatures } S & ::= \langle \rho, \eta \# E \rangle T m(T x) E \\
 \text{Fields } F & ::= T f \text{ in } R \\
 \text{Methods } M & ::= S \{ e \} \\
 \text{RPLs } R & ::= r \mid \rho \mid R:r \mid R:* \\
 \text{Types } T & ::= I \langle T, R, E \rangle \mid C \langle T, R \rangle \mid \tau \langle R \rangle \mid \text{Null} \\
 \text{Effects } E & ::= \emptyset \mid \text{reads } R \mid \text{writes } R \mid \eta \mid E \cup E \\
 \text{Expressions } e & ::= \text{this}.f \mid \text{this}.f=e \mid e.\langle R, E \rangle m(e) \mid v \mid \text{new } T \mid \text{null} \\
 \text{Variables } v & ::= \text{this} \mid x
 \end{array}$$

Fig. 11. Syntax of the core language. $r, I, \tau, \rho, \eta, C, f, m,$ and x are identifiers

Figure 11 gives the syntax for the core language. A program \mathcal{P} consists of region name declarations, interface definitions, class definitions, and an expression to evaluate. An interface \mathcal{I} consists of an interface name I , the interface parameters, and zero or more method signatures. There is one type parameter τ , one region parameter ρ , and one constrained effect parameter $\eta \# E$. The type parameter τ has a region parameter ρ that captures the region argument of the type bound to it. A method signature S specifies a region parameter, a constrained effect parameter, a return type, a method name m , a typed formal parameter x , and an effect.

A class \mathcal{C} consists of a class name C , the class parameters, the interface type being implemented, and the fields and methods of the class. For simplicity we omit class effect parameters; their treatment is identical to interface effect parameters. A field F specifies a type, a field name f , and an RPL. A method specifies a signature and an expression to evaluate. A region path list (RPL) R is a named region r , a region parameter ρ , or an RPL qualified by appending $:r$ or $:*$, where $*$ stands in for any chain of names. A type T instantiates a named interface with a type, region, and effect; or it instantiates a named class with a

type and region; or it instantiates a type parameter with a region; or it is `Null`. `Null` is the type of a null reference. It also functions as a base-case type for type parameter arguments (every other type has its own argument). An effect E is a possibly empty union of read effects, write effects, and effect parameters. An expression e is a field access, field assignment, method invocation, variable, object creation, or null reference. A variable v is `this` or a method parameter x .

4.2 Static Semantics

Environment. We define the static semantics with respect to a static environment Γ , defined as follows:

$$\Gamma ::= \emptyset \mid (v, T) \mid \tau \mid \rho \mid \eta \mid \eta \# E \mid \Gamma \cup \Gamma$$

(v, T) means that variable v has type T ; τ , ρ , or η means that the parameter is in scope; and $\eta \# E$ means that the effect bound to η constrained not to interfere with effect E .

Translation mapping ϕ_T . We define a mapping ϕ_T for translating a type, region, or effect defined in an interface I or class C to its use as a member of a type T instantiating I or C (the *instantiating type*, which must be a class or interface type). It is the context translation described in [5,6], plus effect parameters and type region parameters. Figure 12 gives the key formal rules for interface types; the rules for class types are similar. Note that when the instantiating type has

$\phi_T(T')$	$\phi_T(I\langle T', R, E \rangle) = I\langle \phi_T(T'), \phi_T(R), \phi_T(E) \rangle$
$\phi_T(R)$	$\phi_{I\langle T, R, E \rangle}(\tau\langle R' \rangle) = I\langle T, \phi_{I\langle T, R, E \rangle}(R'), E \rangle$
$\phi_T(E)$	$\phi_{I\langle T, R, E \rangle}(\rho(I)) = R$
	$\phi_{I\langle T, R, E \rangle}(\rho_\tau(I)) = \text{rgn}(T)$ if $T \neq \text{Null}$, else R
	$\phi_T(\text{reads } R) = \text{reads } \phi_T(R)$ $\phi_{I\langle T, R, E \rangle}(\eta(I)) = E$
	$\phi_T(\text{writes } R) = \text{writes } \phi_T(R)$ $\phi_T(E \cup E') = \phi_T(E) \cup \phi_T(E')$

Fig. 12. The translation mapping ϕ_T for interface types (selected rules). $\rho_\tau(T)$, $\rho(T)$, and $\eta(T)$ are the type region parameter, region parameter, and effect parameter of the interface that T instantiates. $\text{rgn}(T)$ is the region argument of T .

a type argument of `Null`, we treat ρ_τ as an alias for ρ . That is because in this simple language, `Null` is the only type with no parameters. In the full language, we support classes and interfaces with no type region parameter (or no type parameter at all), and we disallow bindings of types lacking a region argument to a type parameter with a region parameter.

Program elements. Figure 13 gives the judgments and rules for typing top-level program elements. `METHOD` checks that the method body is well-typed, and that its type and effect agree with the return type and effect specified in the method signature. If a signature with name m also appears in the interface implemented by the enclosing class, then `IMPLEMENT` checks that the types and effects in the method signature agree with the corresponding types and effects in the signature of the implemented interface.

$$\begin{array}{c}
\boxed{\vdash \mathcal{P}} \quad \text{PROGRAM} \quad \frac{\forall I. (\vdash \mathcal{I}) \quad \forall C. (\vdash \mathcal{C}) \quad \emptyset \vdash e : T, E}{\vdash \mathcal{R}^* \mathcal{I}^* \mathcal{C}^* e} \quad \boxed{\vdash \mathcal{I}} \quad \text{INTERFACE} \quad \frac{\Gamma = \tau \cup \rho_\tau \cup \rho \cup \eta \cup \eta \# E \quad \Gamma \vdash E \quad \forall S. (\Gamma \vdash S)}{\vdash \text{interface } I \langle \tau \langle \rho_\tau \rangle, \rho, \eta \# E \rangle \{ S^* \}} \\
\\
\boxed{\vdash \mathcal{C}} \quad \text{CLASS} \quad \frac{\Gamma = \tau \cup \rho_\tau \cup \rho \cup (\text{this}, C \langle \tau \langle \rho_\tau \rangle, \rho \rangle) \quad \Gamma \vdash I \langle T, R, E \rangle \quad \forall F. (\Gamma \vdash F) \quad \forall M. (\Gamma, I \langle T, R, E \rangle \vdash M)}{\vdash \text{class } C \langle \tau \langle \rho_\tau \rangle, \rho \rangle \text{ implements } I \langle T, R, E \rangle \{ F^* M^* \}} \quad \boxed{\Gamma \vdash F} \quad \text{FIELD} \quad \frac{\Gamma \vdash T \quad \Gamma \vdash R}{\Gamma \vdash T f \text{ in } R} \\
\\
\boxed{\Gamma \vdash S} \quad \text{SIGNATURE} \quad \frac{\Gamma' = \Gamma \cup \rho \cup \eta \cup \eta \# E \quad \Gamma' \vdash T \quad \Gamma' \vdash T' \quad \Gamma' \vdash E \quad \Gamma' \vdash E'}{\Gamma \vdash \langle \rho, \eta \# E \rangle T \ m(T' \ x) \ E'} \\
\\
\boxed{\Gamma, T \vdash M} \quad \text{METHOD} \quad \frac{S = \langle \rho, \eta \# E \rangle T \ m(T' \ x) \ E' \quad \Gamma \vdash S \quad \Gamma' = \Gamma \cup \rho \cup \eta \cup \eta \# E \cup (x, T') \quad \Gamma' \vdash e : T_e, E_e \quad \Gamma' \vdash T_e \preceq T \quad \Gamma' \vdash E_e \subseteq E' \quad m \in \text{Dom}(S(I)) \Rightarrow \Gamma, I \langle T'', R, E \rangle \vdash S \preceq S(I)(m)}{\Gamma, I \langle T'', R, E'' \rangle \vdash S \{ e \}} \\
\\
\boxed{\Gamma, T \vdash S \preceq S'} \quad \text{IMPLEMENT} \quad \frac{\sigma = [\rho_2 \leftarrow \rho_1][\eta_2 \leftarrow \eta_1] \quad \Gamma \vdash \sigma(\phi_T(E_2)) \subseteq E_1 \quad \Gamma \vdash T_1 \preceq \sigma(\phi_T(T_2)) \quad \Gamma \vdash \sigma(\phi_T(T'_2)) \preceq T'_1 \quad \Gamma \vdash E'_1 \subseteq \sigma(\phi_T(E'_2))}{\Gamma, T \vdash \langle \rho_1, \eta_1 \# E_1 \rangle T_1 \ m(T'_1 \ x) \ E'_1 \preceq \langle \rho_2, \eta_2 \# E_2 \rangle T_2 \ m(T'_2 \ x) \ E'_2}
\end{array}$$

Fig. 13. Typing of program elements. $S(I)(m)$ is the signature S named m in the definition of I .

$$\begin{array}{c}
\boxed{\Gamma \vdash R \subseteq R'} \quad \text{INCLUDE-REFL} \quad \frac{}{\Gamma \vdash R \subseteq R} \quad \boxed{\Gamma \vdash R \subseteq R'} \quad \text{INCLUDE-TRANS} \quad \frac{\Gamma \vdash R \subseteq R' \quad \Gamma \vdash R' \subseteq R''}{\Gamma \vdash R \subseteq R''} \quad \boxed{\Gamma \vdash R \subseteq R' : * } \quad \text{INCLUDE-REC} \quad \frac{}{\Gamma \vdash R : r \subseteq R' : * } \quad \boxed{\Gamma \vdash R \subseteq R' : * } \quad \text{INCLUDE-PREF} \quad \frac{}{\Gamma \vdash R \subseteq R' : * } \\
\\
\boxed{\Gamma \vdash R \# R'} \quad \text{DISJOINT-NAMES} \quad \frac{r \neq r'}{\Gamma \vdash r : * \# r' : * } \quad \boxed{\Gamma \vdash R \subseteq R' \quad \Gamma \vdash R'' \subseteq R'''} \quad \text{DISJOINT-INCLUDE} \quad \frac{}{\Gamma \vdash R \# R''}
\end{array}$$

Fig. 14. Inclusion and disjointness of RPLs

RPLs. Figure 14 gives the rules for inclusion and disjointness of RPLs. We use a relevant subset of the rules described in 6. R is included in R' if R and R' are lexically identical, or if $R' = R'' : *$, and R'' is a prefix of R . For example, $r : * : r'$ is included in $r : *$. Two RPLs are disjoint if they both start with different names r , or if each is included in another RPL, such that the two including RPLs are disjoint. As in 6, inclusion and disjointness of RPLs correspond to inclusion and disjointness of the sets of regions (chains of names r) represented by the RPLs.

Types. Figure 15 gives the rules for checking types. TYPE-INTERFACE checks the disjointness requirement for the effect argument to an interface type. $\Gamma \vdash T \preceq T'$ means that T is a subtype of T' , and $\Gamma \vdash T \subseteq T'$ means that T and T' are the same type, except that the region and effect arguments are related by inclusion. The inclusion relation \subseteq implies subtyping (rule SUBTYPE-INCLUDE), but not vice versa. Note that it would not be sound to put $\Gamma \vdash T \preceq T'$ in the condition of INCLUDE-INTERFACE or INCLUDE-CLASS, for the same reason that it is not sound to treat $C \langle C' \rangle$ as a subtype of $C \langle \text{Object} \rangle$ in ordinary Java 17. It is

<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">$\Gamma \vdash T$</div> $\frac{\text{TYPE-INTERFACE} \quad \text{interface } I \langle \tau \langle \rho_r \rangle, \rho, \eta \# E' \rangle \{ S^* \} \in \mathcal{P} \quad \Gamma \vdash T \quad \Gamma \vdash R \quad \Gamma \vdash E \quad \Gamma \vdash E \# \phi_{I \langle T, R, E \rangle}(E')}{\Gamma \vdash I \langle T, R, E \rangle}$	$\frac{\text{TYPE-CLASS} \quad \text{defined}(C) \quad \Gamma \vdash T \quad \Gamma \vdash R}{\Gamma \vdash C \langle T, R \rangle}$	$\frac{\text{TYPE-PARAM} \quad \tau \in \Gamma \quad \Gamma \vdash R}{\Gamma \vdash \tau \langle R \rangle}$
<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">$\Gamma \vdash T \preceq T'$</div> $\frac{\text{SUBTYPE-INCLUDE} \quad \Gamma \vdash T \subseteq T' \quad \Gamma \vdash T \preceq T'}{\Gamma \vdash T \preceq T'}$	$\frac{\text{SUBTYPE-IMPLEMENT} \quad \text{class } C \langle \tau \langle \rho_r \rangle, \rho \rangle \text{ implements } I \langle T, R, E \rangle \{ F^* M^* \} \in \mathcal{P}}{\Gamma \vdash C \langle T', R' \rangle \preceq \phi_{C \langle T', R' \rangle}(I \langle T, R, E \rangle)}$	
<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">$\Gamma \vdash T \subseteq T'$</div> $\frac{\text{INCLUDE-INTERFACE} \quad \Gamma \vdash T \subseteq T' \quad \Gamma \vdash R \subseteq R' \quad \Gamma \vdash E \subseteq E'}{\Gamma \vdash I \langle T, R, E \rangle \subseteq I \langle T', R', E' \rangle}$	$\frac{\text{INCLUDE-CLASS} \quad \Gamma \vdash T \subseteq T' \quad \Gamma \vdash R \subseteq R'}{\Gamma \vdash C \langle T, R \rangle \subseteq C \langle T', R' \rangle}$	$\frac{\text{INCLUDE-PARAM} \quad \Gamma \vdash R \subseteq R'}{\Gamma \vdash \tau \langle R \rangle \subseteq \tau \langle R' \rangle}$

Fig. 15. Types (selected rules). $\text{defined}(C)$ means that class C is defined in the program.

sound, however, to make inclusion a condition of subtyping, because we capture regions and effects as discussed below.

<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">$\Gamma \vdash E \subseteq E'$</div> $\frac{\text{SE-EMPTY} \quad \Gamma \vdash \emptyset \subseteq E}{\Gamma \vdash \emptyset \subseteq E}$	$\frac{\text{SE-READS} \quad \Gamma \vdash R \subseteq R'}{\Gamma \vdash \text{reads } R \subseteq \text{reads } R'}$	$\frac{\text{SE-WRITES} \quad \Gamma \vdash R \subseteq R'}{\Gamma \vdash \text{writes } R \subseteq \text{writes } R'}$
$\frac{\text{SE-READS-WRITES} \quad \Gamma \vdash R \subseteq R'}{\Gamma \vdash \text{reads } R \subseteq \text{writes } R'}$	$\frac{\text{SE-UNION-1} \quad \Gamma \vdash E \subseteq E'}{\Gamma \vdash E \subseteq E' \cup E''}$	$\frac{\text{SE-UNION-2} \quad \Gamma \vdash E' \subseteq E \quad \Gamma \vdash E'' \subseteq E}{\Gamma \vdash E' \cup E'' \subseteq E}$
<div style="border: 1px solid black; padding: 2px; display: inline-block; margin-bottom: 5px;">$\Gamma \vdash E \# E'$</div> $\frac{\text{NI-EMPTY} \quad \Gamma \vdash \emptyset \# E}{\Gamma \vdash \emptyset \# E}$	$\frac{\text{NI-READS}}{\Gamma \vdash \text{reads } R \# \text{reads } R'}$	$\frac{\text{NI-WRITES} \quad \Gamma \vdash R \# R'}{\Gamma \vdash \text{writes } R \# \text{writes } R'}$
$\frac{\text{NI-UNION} \quad \Gamma \vdash E \# E'' \quad \Gamma \vdash E' \# E''}{\Gamma \vdash E \cup E' \# E''}$	$\frac{\text{NI-PARAM} \quad \eta \# E \in \Gamma}{\Gamma \vdash \eta \# E}$	$\frac{\text{NI-INCLUDE} \quad \Gamma \vdash E'' \subseteq E \quad \Gamma \vdash E''' \subseteq E'}{\Gamma \vdash E'' \# E'''}$

Fig. 16. Subeffects and disjoint effects

Effects. Figure 16 gives the relevant rules for subeffects and noninterfering effects. For subeffects, reads effects on R cover reads of R' if R includes R' , and write effects on R cover both reads and writes of R' . For noninterfering effects, read effects are always noninterfering, writes effects are noninterfering if the regions are disjoint, and parametric effects are disjoint if disjointness is specified in a constraint.

Expressions. Figure 17 gives the rules for typing expressions. INVOKE translates types and effects in the method signature by (1) substituting for the method region and effect arguments; and (2) applying the translation mapping ϕ_T defined above. Here T is the type of the dispatch expression, after capturing its region and effect arguments. As discussed in Section 3.2, we must capture all partially specified RPLs and all effects. We capture an RPL or effect by replacing it with a fresh parameter, and adding the parameter to the environment. We do this for the RPL and effect arguments of T , and recursively for the type argument of T . The formal rules for this procedure are straightforward and are stated in full in 5.

$$\begin{array}{c}
\boxed{\Gamma \vdash e : T, E} \quad \text{ACCESS} \quad \frac{(\text{this}, C \langle \tau \langle \rho \tau \rangle, \rho \rangle) \in \Gamma \quad \mathcal{F}(C)(f) = T \text{ f in } R}{\Gamma \vdash \text{this.f} : T, \text{reads } R} \quad \text{ASSIGN} \quad \frac{(\text{this}, C \langle \tau \langle \rho \tau \rangle, \rho \rangle) \in \Gamma \quad \Gamma \vdash e : T, E \quad \mathcal{F}(C)(f) = T' \text{ f in } R \quad \Gamma \vdash T \preceq T'}{\Gamma \vdash \text{this.f} = e : T, E \cup \text{writes } R} \\
\text{INVOKe} \quad \frac{\text{VARIABLE} \quad \frac{(v, T) \in \Gamma \quad \Gamma \vdash v : T, \emptyset}{\Gamma \vdash e_1 : T_1, E_1} \quad \Gamma \vdash e_2 : T_2, E_2 \quad \mathcal{S}(T_1)(m) = \langle \rho, \eta \# E_3 \rangle T_3 \quad m(T_4 \ x) \ E_4 \quad \sigma = [\rho \leftarrow R][\eta \leftarrow E_5]}{\Gamma \vdash E_5 \# \sigma(\phi_{T_1}(E_3)) \quad \Gamma \vdash \text{capt}(T_1) = (T_c, \Gamma_c) \quad \Gamma_c \vdash T_2 \preceq \sigma(\phi_{T_c}(T_4))}{\Gamma \vdash e_1 \cdot \langle R, E_5 \rangle m(e_2) : \sigma(\phi_{T_1}(T_3)), E_1 \cup E_2 \cup \sigma(\phi_{T_1}(E_4))} \\
\text{NEW} \quad \frac{\Gamma \vdash C \langle T, R \rangle}{\Gamma \vdash \text{new } C \langle T, R \rangle : C \langle T, R \rangle, \emptyset} \quad \text{NULL} \quad \frac{}{\Gamma \vdash \text{null} : \text{Null}, \emptyset}
\end{array}$$

Fig. 17. Expressions. $\mathcal{F}(C)(f)$ means field f declared in class C . $\mathcal{S}(T)$ means $\mathcal{S}(I)$ or $\mathcal{S}(C)$, corresponding to the interface or class named in T . $\Gamma \vdash \text{capt}(T) = (T', \Gamma')$ means that capturing type T in environment Γ yields type T' and environment Γ' .

4.3 Dynamic Semantics

$$\begin{array}{c}
\boxed{(e, \Sigma, H) \rightarrow (o, H', E)} \quad \text{DYN-ACCESS} \quad \frac{(\text{this}, o) \in \Sigma \quad H(o) = (O, C \langle T, R \rangle) \quad \mathcal{F}(C)(f) = T' \text{ f in } R'}{(\text{this.f}, \Sigma, H) \rightarrow (O(f), H, \text{reads } \phi_{\Sigma, H}(R'))} \\
\text{DYN-ASSIGN} \quad \frac{(e, \Sigma, H) \rightarrow (o, H', E) \quad (\text{this}, o') \in \Sigma \quad H'(o') = (O, C \langle T, R \rangle) \quad \mathcal{F}(C)(m) = T' \text{ f in } R'}{(\text{this.f} = e, \Sigma, H) \rightarrow (o, H' \{o' \mapsto (O[f \mapsto o], C \langle T, R \rangle)\}, E \cup \text{writes } \phi_{\Sigma, H}(R'))} \\
\text{DYN-INVOKe} \quad \frac{(e_1, \Sigma, H_1) \rightarrow (o_1, H_2, E_2) \quad (e_2, \Sigma, H_2) \rightarrow (o_2, H_3, E_3) \quad H_3(o_1) = (O, C \langle T_1, R' \rangle) \quad \mathcal{M}(C)(m) = \langle \rho, \eta \# E_4 \rangle T_2 \quad m(T_3 \ x) \ E_5 \ \{e_3\}}{\Sigma' = (\text{this}, o_1) \cup (x, o_2) \cup (\rho, \phi_{\Sigma, H}(R)) \cup (\eta, \phi_{\Sigma, H}(E_1)) \quad (e_3, \Sigma', H_3) \rightarrow (o_3, H_4, E_6)} \\
\frac{}{(e_1 \cdot \langle R, E_1 \rangle m(e_2), \Sigma, H_1) \rightarrow (o_3, H_4, E_2 \cup E_3 \cup E_6)} \\
\text{DYN-VARIABLE} \quad \frac{(z, o) \in \Sigma}{(z, \Sigma, H) \rightarrow (o, H, \emptyset)} \quad \text{DYN-NEW} \quad \frac{o \notin \text{Dom}(H) \quad H' = H \cup o \mapsto (\text{new}(C), \phi_{\Sigma, H}(C \langle T, R \rangle))}{(\text{new } C \langle T, R \rangle, \Sigma, H) \rightarrow (o, H', \emptyset)}
\end{array}$$

Fig. 18. Program evaluation. $f[a \mapsto b]$ denotes the function identical to f everywhere on its domain, except that it maps a to b . $\text{new}(C)$ is the function taking each field of class C to null . The translation function $\phi_{\Sigma, H}$ does the following: (1) it substitutes actual regions and effects for parameters as specified by the bindings in Σ ; (2) if $(\text{this}, o) \in \Sigma$ and $(O, T) \in H$, it applies ϕ_T . $\mathcal{M}(C)(m)$ denotes the method named m in the definition of class C .

We give a large-step semantics for program execution, using the transition relation $(e, \Sigma, H) \rightarrow (o, H', E)$. e is a program expression. The dynamic environment Σ maps variables v to object references o , region parameters ρ to regions R , and effect parameters η to effects E :

$$\Sigma ::= (v, o) \mid (\rho, R) \mid (\eta, E)$$

The heap H is a partial function from object references o to pairs $(O, C \langle T, R \rangle)$, where O is an object, and $C \langle T, R \rangle$ is the type of O :

$$H ::= \text{null} \mid o \mapsto (O, C \langle T, R \rangle) \mid H \cup H$$

`null` is a special reference that is in $\text{Dom}(H)$ but does not map to an object. Attempting to access a field of `null` causes execution to fail. An object O is a mapping from field names f to object references o :

$$O ::= \emptyset \mid f \mapsto o \mid O \cup O$$

The effect E collects the effect of the evaluation. A program evaluates to reference o with heap H and effect E if its main expression is e and $(e, \text{null}, \emptyset) \rightarrow (o, H, E)$, according to the rules shown in Figure 18. The rules describe a standard semantics for an object-oriented language, except that we bind actual regions and effects to method parameters in rule INVOKE, and we accumulate the effects of every expression evaluation.

4.4 Valid Execution State

To state the soundness results, we need to define valid heaps, environments, and execution states.

$$\begin{array}{c}
 \boxed{\vdash H} \quad \text{HEAP} \quad \boxed{H \vdash H'} \quad \text{HEAP-NULL} \quad \text{HEAP-OBJECT} \quad \text{HEAP-UNION} \\
 \frac{H \vdash H}{\vdash H} \quad \frac{}{H \vdash \text{null}} \quad \frac{H \vdash (O, T)}{H \vdash o \mapsto (O, T)} \quad \frac{H \vdash H' \quad H \vdash H''}{H \vdash H' \cup H''} \\
 \\
 \boxed{H \vdash o : T} \quad \text{TYPE-OBJECT} \quad \text{TYPE-NULL} \\
 \frac{o \mapsto (O, T) \in H}{H \vdash o : T} \quad \frac{}{H \vdash \text{null} : \text{Null}} \\
 \\
 \text{OBJECT} \\
 \boxed{H \vdash (O, T)} \quad \frac{\forall (f \in \text{Dom}(\mathcal{F}(C))). (\mathcal{F}(C)(f) = T' \ f \ \text{in} \ R' \wedge H \vdash O(f) : T'' \wedge \emptyset \vdash T'' \preceq \phi_{C \langle T, R \rangle}(T')) \quad \emptyset \vdash C \langle T, R \rangle}{H \vdash (O, C \langle T, R \rangle)}
 \end{array}$$

Fig. 19. Well-typed heaps

Heaps. Figure 19 gives the rules for typing heaps. A heap is valid if its elements are valid. An object-type pair (O, T) is valid if (1) T is a valid type; and (2) for every field f in $\mathcal{F}(C)$, $O(f)$ is defined, and its type is a subtype of the static type of f , after translation via ϕ_T . At runtime we check types in the empty environment, because all parameters have been substituted away.

Environments. A static environment is valid ($\vdash \Gamma$) if it binds variables to valid types, and if the effects named in the constraints are valid. A dynamic environment is valid ($H \vdash \Sigma$) if it binds variables to valid object references and parameters to valid regions and effects. We omit the formal rules for valid environments, but they are straightforward and are stated in full in 5.

Execution state. Figure 20 gives the rules for a valid execution state (e, Σ, H) , with respect to the environment Γ that typed e in the static semantics. The rules state that H , Σ , and Γ are valid; e is well typed in Γ ; and Σ instantiates Γ ($H \vdash \Sigma \preceq \Gamma$). That means the types of the variable bindings in Σ and Γ match, and the bindings in Σ obey the disjointness constraints specified by Γ .

$$\begin{array}{c}
\boxed{\Gamma \vdash (e, \Sigma, H) : T, E} \quad \text{STATE} \quad \frac{\begin{array}{c} \vdash \Gamma \quad \vdash H \quad H \vdash \Sigma \\ H \vdash \Sigma \preceq \Gamma \quad \Gamma \vdash e : T, E \end{array}}{\Gamma \vdash (e, \Sigma, H) : T, E} \quad \boxed{H \vdash \Sigma \preceq \Gamma} \quad \text{INSTANTIATE} \quad \frac{\Sigma, H \vdash \Sigma \preceq \Gamma}{H \vdash \Sigma \preceq \Gamma} \\
\boxed{\Sigma, H \vdash \Sigma' \preceq \Gamma} \quad \text{INST-VAR} \quad \frac{H \vdash o : T \quad \emptyset \vdash T \preceq \phi_{\Sigma, H}(T')}{\Sigma, H \vdash (z, o) \preceq (z, T')} \quad \text{INST-CONSTRAINT} \quad \frac{\emptyset \vdash \phi_{\Sigma, H}(\eta) \# \phi_{\Sigma, H}(E)}{\Sigma, H \vdash \emptyset \preceq \eta \# E}
\end{array}$$

Fig. 20. Valid execution state (selected rules)

4.5 Soundness Results

Preservation of type and effect. The first soundness result states that the static types and effects computed according to Figure 17 approximate the dynamic types and effects produced by execution according to Figure 18. More precisely, if we evaluate e to o starting in a valid execution state, then the resulting heap is valid; o is well typed, and its type is a subtype of the static type of e ; and the resulting effect is valid and a subeffect of the static effect of e .

Theorem 1 (Preservation of type and effect). *If $\vdash \mathcal{P}$ and $\Gamma \vdash (e, \Sigma, H) : T_s, E_s$ and $(e, \Sigma, H) \rightarrow (o, H', E)$, then (a) $\vdash H'$; (b) $H' \vdash o : T$; (c) $\emptyset \vdash T \preceq \phi_{\Sigma, H'}(T_s)$; (d) $\emptyset \vdash E$; and (e) $\emptyset \vdash E \subseteq \phi_{\Sigma, H'}(E_s)$.*

The proof, stated in full in 5, is by induction on the structure of e , showing the result for each of the rules given in Figure 18. For all rules but INVOKE, the result follows straightforwardly from the assumptions and the induction hypothesis. For INVOKE, we must show two facts: first, that the dynamic environment in which the method body is executed instantiates the static environment in which we typed the method; and second, that the preservation properties are preserved when we translate back to the environment in which we typed the method invocation. Both facts are proved by keeping careful track of the substitutions that occur in translating from one static environment to another, and in translating from static to dynamic environments.

Here it helps the proof that $C \langle E \rangle$ is a subtype of $C \langle E' \rangle$ if E is a subeffect of E' , as discussed in Section 3.2. If we required $E = E'$ in the subtype judgment, then we would not be able to conclude that T a subtype of T' implies $\phi_T(T'')$ a subtype of $\phi_{T'}(T'')$. In that case, to ensure sound subtyping for method invocations, we would need to introduce some ad-hoc restrictions on the use of type region parameters in effects.

Noninterference. The second soundness result states that the static noninterference judgment for expressions is sound: if two expressions have statically noninterfering effects, then the execution of the two expressions is noninterfering at runtime.

Theorem 2 (Noninterference). *If $\vdash \mathcal{P}$ and $\Gamma \vdash (e, \Sigma, H) : T_s, E_s$ and $\Gamma \vdash (e', \Sigma, H') : T'_s, E'_s$ and $\Gamma \vdash E_s \# E'_s$ and $(e, \Sigma, H) \rightarrow (o, H', E)$ and $(e', \Sigma, H') \rightarrow (o', H'', E')$, then there are no conflicting memory operations in the evaluations of e and e' .*

“Conflicting accesses” means a pair of operations on the same memory location, one or both of which is a write. Again the proof is stated in full in [5]. Theorem 11 says that the static effects contain the dynamic effects, so it suffices to show that conflicting accesses produce interfering effects. But this is straightforward from (1) the way that the rules in Figure 13 record effects; and (2) the definition of noninterfering effects in Figure 16.

5 Evaluation

We have evaluated the techniques discussed above with two goals in mind. First, can we use the techniques to write realistic frameworks and user programs? Do any additional issues arise in real frameworks or user code? Second, what is the complexity and annotation overhead of using the techniques to write framework APIs and client code?

We extended the DPJ compiler [6, 7] to support the new language features discussed in Sections 3 and 4. Then we studied how to (1) use our techniques to write generic array, tree, and pipeline frameworks; and (2) use the frameworks to write three parallel codes: a Monte Carlo simulation algorithm, a Barnes-Hut n-body computation using a tree to partition physical space, and radix sort expressed as a pipeline. We chose these three algorithms because they exemplify different styles of parallelism.

5.1 DPJ Frameworks

Array. We wrote a framework **DPJDisjointArray** with an API similar to a subset of Java’s `ParallelArray` [1]. Our API supports creating an array, mapping one array to another with a user-supplied element mapping function, and reducing the array to a single element with a user-supplied binary reduction method (i.e., that reduces two elements into one). For the array creation and mapping interfaces, we used exactly the techniques discussed in Section 3. For the reduction operation, we had to solve the following problem: the user-supplied binary reduction method might violate disjointness by, e.g., storing one of its argument objects into a field of the other. To prevent that, we parameterized each of the arguments with a separate method region parameter, as follows:

```
public interface Reducer<type T<region R>, effect E> {
    public <region R1,R2>T<R1> op(T<R1> a, T<R2> b) writes R1,R2 effect E;
}
```

Tree. We wrote a framework **DPJDisjointTree** that provides a tree with a user-specified branching factor. The tree stores a data object of generic type `T` in each node. The API supports building a tree by inserting bodies from the root and doing a recursive parallel postorder traversal over the tree. The build method takes a user-supplied `index` function that computes which of the children of a particular node to follow next when inserting an object in the subtree rooted at that node. The postorder visitor takes a user-supplied `visit` method. The

input to the method (furnished by the framework implementation) consists of the data object at the current node and an `ArrayList` of result objects produced from visiting the children (or `null` if the current node is a leaf). The output is a result object for the current node. Again we use two region parameters to ensure that the `visit` method preserves disjointness for the data objects.

Pipeline. We implemented a framework **DPJPipeline** that represents data flowing through a series of pipeline stages, each of which applies some operation to the data. Following the Threading Building Blocks (TBB) library [28] and the StreamIt language [34], we call the operation applied by each stage a *filter*. Each data element flows sequentially through the stages, but different stages can apply their filters to different elements at the same time, creating pipeline parallelism.

The API provides two interfaces for the user to implement: a filter and a factory method for creating a filter. Method region parameters on the factory methods ensure that each filter and each element is a freshly-created object. The filter interface provides an operation method for the user to override. Using method region parameters and constrained effect variables, as in the other examples, the API ensures that the user-defined filter operation is limited to updating the regions of the data object and the filter state, and doing any noninterfering effects on other state. In particular, the filter operation may not update data operated on by a concurrent filter, or a different filter.

5.2 Client Code

Monte Carlo Simulation. We studied the Monte Carlo simulation benchmark from the Java Grande suite [30]. The computation contains three parallelizable loops: the first one creates task objects, the second one iterates over the objects to compute a return rate for each one, and the third one reduces the return rates into a cumulative average. We parallelized all three loops using `DPJDisjointArray`. The first two loops were straightforward to parallelize with the mapping operation. For the third loop, we wrote a binary reduction method that takes two objects produced by the second stage, reads the accumulated sum from both, adds them, stores the result in the first one, and returns it. We could also have created a new object and returned it, but that would be less efficient.

Barnes-Hut Center of Mass. Next we studied the Barnes-Hut n-body simulation [29], which uses an octree (eight-ary tree) to represent three-dimensional space hierarchically, storing the bodies in the leaves. We focused on the center-of-mass computation, which traverses the tree recursively in parallel and, for each node, computes and stores the center of mass of the subtree rooted at that node. It would be straightforward to parallelize the force computation using the same array-based techniques that we used for Monte Carlo.

We wrote a program that builds a tree and performs a center of mass computation for a binary tree computation in one-dimensional space. That simplified the computation, while retaining the algorithm structure. To do this, we instantiated `DPJDisjointTree` with a `Node` class that has subclasses `Cell` for the inner

node data and `Body` for the leaf data, similarly to the original and SPLASH versions [29]. Then, studying the original algorithm, we put the logic for creating the tree into the user-supplied `index` function and the logic for computing and storing the center of mass into the user-supplied `visit` function.

Radix Sort. We wrote a pipelined version of radix sort that is directly modeled after the StreamIt RadixSort benchmark [34]. The first stage produces a stream of arrays to sort, and the successive stages each sort the arrays on a different radix, with the radix recorded in the `Filter` object as `final` variable (so reading it produces no effect). Each sort stage also stores two temporary arrays as persistent mutable data in the filter region (such that accessing the arrays produces an effect on the filter region). When an array enters a sort stage, the filter for that stage adds each array element to one of the temporary arrays, depending on whether the element has a 0 or 1 at the bit position corresponding to the radix for that filter. The filter then copies all the 0 elements followed by all the 1 elements back into the original array, and passes it to the next stage.

5.3 Discussion of Evaluation Results

Expressivity. We were able to use the techniques discussed in Section 3 to write realistic parallel frameworks, with no significant additional challenges. Getting the region and effect annotations correct for the APIs did require some careful thought. However, all the APIs have a similar pattern; once we mastered that pattern, writing the APIs was straightforward.

Table 1 summarizes the effect annotation counts for the framework code. The leftmost data column shows the annotated over the total source lines of code (SLOC), counted with `sloccount`. From the left, the other columns show the number of class (including interface) definitions, class region and effect parameters, class region and effect constraints, region and effect arguments to types, method definitions, method effect summaries, method region and effect parameters, method region and effect constraints, and region and effect arguments to methods. For arguments to class types, the denominator is the total number of types appearing in the program; and for arguments to methods, the denominator is the total number of method invocations. As expected, the annotations are nontrivial; this is simply a cost of the safety guarantee we provide. We believe that production frameworks would have a higher ratio of internal to API code than our simple frameworks do, so the relative annotation overhead would be lower in practice. Further, some type and effect annotations could be inferred. In particular [36] shows how to infer method effect summaries for DPJ as described in [6]; this approach could be extended to inferring arguments to region and effect parameters in types and method invocations.

Table 1. Annotation counts for the framework code

	SLOC	Classes				Methods				
		Defs	Params	Constr.	Args	Defs	Summ.	Params	Constr.	Args
Array	41/97	12	21	0	10/88	20	11	7	4	1/21
Tree	61/169	11	19	0	32/100	18	16	6	2	4/42
Pipeline	35/112	8	9	1	14/44	19	18	2	0	2/28

Table 2. Annotation counts for the client code

	SLOC	Classes				Methods				
		Defs	Params	Constr.	Args	Defs	Summ.	Params	Constr.	Args
Monte Carlo	236/1389	21	10	0	90/492	195	136	8	0	3/350
Spatial Tree	55/172	6	5	0	42/90	10	7	4	0	3/45
Radix Sort	31/102	6	3	0	36/46	11	6	4	0	0/13

Framework Client Experience. Table 2 shows the annotation counts for the client code, with the same layout as Table 1. Overall, the annotation burden is less than for the framework code. As in 6, most of the annotations are method effect summaries and region arguments to types. In the client codes, the arguments to effect variables were simple: either `pure` or one or two read effects. As expected, there were no effect constraints in the client code. Again, type and effect inference 36 could reduce the annotation burden.

It is also instructive to compare the client experience to DPJ as presented in 6.7. In 6, we wrote Monte Carlo using an index-parameterized array for the first two loops; for the third loop, we encapsulated the reduction sum in a method implemented with locks and declared that method `commutative`. This is not attractive because it puts the burden of writing low-level synchronization code on the application developer. To write the Barnes-Hut center of mass computation using the techniques shown in 6, each tree node would need a distinct type. Because the destination node of a body is not known at the time the body is originally created, we would have to recopy the bodies on insertion into the tree; this is similar to the swapping example discussed in Section 2. This approach works, but it adds overhead. For pipelined radix sort, we could write this program using the features presented in 7 for safe nondeterminism, but we would need to use low-level synchronization techniques in the client program, and we would not get the framework encapsulation or any determinism guarantee.

Overall, the advantages of the framework approach are (1) simplifying the DPJ types exposed to the client, by avoiding index parameterized arrays or recursive types; (2) eliminating low-level code for common patterns such as reductions; (3) supporting operations such as reshuffling that the type system prohibits; and (4) extending the language with more flexible parallel control idioms. On the other hand, the non-framework DPJ code is closer in structure to the original sequential program. This last point is not specific to our work, but is a general issue with frameworks.

6 Related Work

Effects. The seminal work on types and effects for concurrency is FX 19,24, which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. Later work added effects to object-oriented languages 18,21. DPJ 6.7 builds upon this work to provide an expressive type and effect system for deterministic-by-default parallelism. None of this work teaches how to write a framework API for safe parallelism using disjoint data structures. Nor does it support mechanisms such as effect constraints and type region parameters that are necessary for generic frameworks.

Several sophisticated effect systems are based on *object ownership* [9,12,22,23]. There are many variants, all unified by the idea that objects define groupings of data on the heap (which we call regions). DPJ is similar in that it uses regions to group data on the heap, but it is different in that a region is specified by an RPL, which is primarily a sequence of declared names like $A:B:C$. DPJ as described in [6] incorporates a notion of ownership by allowing an object to appear first in an RPL (as in $o:A:B:C$). Though useful for some parallel patterns, this form is not used in the present work. Ownership domains [2,31] provide an alternative way to combine declared names with owner objects.

Linear Types. Linear types [37] allow in-place updates while preserving the semantic guarantees of pure functional programming. However, linear types prohibit reference aliasing, making many common patterns of imperative programming awkward or impossible.

Several researchers have worked to make linear types less restrictive while maintaining meaningful guarantees. Fähndrich and DeLine [16] introduced *adoption and focus* to create aliases of a linear reference with a limited lifetime. Clarke and Wrigstad [13] have observed that *external uniqueness* — the property that every object has at most one reference to it located outside its containing data structure — can express important patterns. Boyland and others [8,33] have used *fractional permissions* to enforce linearity of write references, while allowing sharing of read-only references.

Our idea of *disjoint data structures* is related to these mechanisms, but also different from all of them. Our insight is that for parallel traversals over the elements of a data structure, all we care about is whether the elements have different regions in their types. This implies that the elements are distinct objects, but it does not preclude aliasing with other references in the program. DPJ's indexed parameterized arrays [6] provide disjoint regions, but they do so by making the regions explicit in user code, thereby preventing reference swapping as discussed in Section 2.

Enforcing API Contracts. The Eiffel language [35] introduced *design by contract*, which uses preconditions and postconditions to specify interaction between classes. Spec# [4] and the Java Modeling Language (JML) [20] provide ways to write design-by-contract specifications for C# and Java; the specifications can be checked with a combination of static verification and online checking.

Design-by-contract ideas have been applied to concurrent programming languages. Meyer's Systematic Concurrent Object-Oriented Programming (SCOOP) concurrent programming model [25] is based on Eiffel. The Fortress programming language [32] provides a way to write assertions at interface boundaries that can be checked at runtime. X10 [11] has a sophisticated dependent type system that can specify and check interface assertions, also supported with runtime checking. None of this work addresses parallel noninterference or safe frameworks for shared memory parallelism.

Separation Logic. Recent work on separation logic (SL) [14, 15] shows how to specify abstractions such as barriers, locks, and sets and verify separately that (1) programs using the abstractions are correct and (2) the abstractions are correctly implemented. While similar in spirit to ours, this work does not consider the abstractions we have studied, including data structures containing references to disjoint mutable objects, frameworks with internal parallelism, and frameworks applying methods with unknown effects. The technical mechanisms are also very different (SL-based program verification vs. types and effects). Compared to SL, our effect system is better integrated with the language, easier to use, and amenable to lightweight checking; however, it is probably also less powerful. More complex logics such as SL could be used to prove that a framework implementation satisfies the properties stated in Section 3.3.

Type Constructors. Type constructors are well known in functional languages like Haskell. Recently type constructors have been applied to object-oriented languages [3, 26]. Standard type constructors have no notion of region parameters or effects. Further, we exploit the fact that a parametric type implicitly provides a type constructor: in our language one can define a class `List<type T<region R>>` and use `T` either as the type T bound to the type argument of `List`, or as the type constructor that results from ignoring the binding to the first region parameter in T . This can be viewed as syntactic sugar for a more standard approach, where one would specify the region R as a separate class parameter. Ownership Generic Java [27] uses a similar approach in specifying type bounds, but it does not have any notion of effects or support type constructors.

7 Conclusion

We have shown how to use an effect system with polymorphic effects and type constructors to write a generic framework API that enables sound reasoning about its uses. The framework internals can be checked once, and then the compiler can guarantee noninterference for any user program written using the framework. As future work, we would like to explore ways to formally verify properties of the framework implementation that DPJ cannot prove.

References

1. <http://gee.cs.oswego.edu/dl/jsr166/dist/extra166ydocs/index.html?extra166y/package-tree.html>
2. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
3. Altherr, P., Cremet, V.: Adding type constructor parameterization to Java. In: Formal Techniques for Java-like Programs, FTFJP (2007)

4. Barnett, M., et al.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
5. Bocchino, R.: An Effect System and Language for Deterministic-by-Default Parallel Programming. PhD thesis, Univ. of Illinois, Urbana-Champaign, IL (2010)
6. Bocchino, R., et al.: A type and effect system for deterministic parallel Java. In: OOPSLA (2009)
7. Bocchino, R., et al.: Safe nondeterminism in a deterministic-by-default parallel language. In: POPL (2011)
8. Boyland, J.: In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, Springer, Heidelberg (2003)
9. Cameron, N., et al.: Multiple ownership. In: OOPSLA (2007)
10. Cameron, N., Gairing, M., Bateni, M.: A model for Java with wildcards. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 2–26. Springer, Heidelberg (2008)
11. Charles, P., et al.: X10: An object-oriented approach to non-uniform cluster computing. In: OOPSLA (2005)
12. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA (2002)
13. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 176–201. Springer, Heidelberg (2003)
14. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D’Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
15. Dodds, M., et al.: Modular reasoning for deterministic parallelism. In: POPL (2011)
16. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: PLDI (2002)
17. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley Longman, Amsterdam (2005)
18. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: Liu, H. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 205–229. Springer, Heidelberg (1999)
19. Hammel, R., Gifford, D.: FX-87 performance measurements: Dataflow implementation. Technical Report MIT/LCS/TR-421 (1988)
20. Leavens, G., et al.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes (2006)
21. Leino, K., et al.: Using data groups to specify and check side effects. In: PLDI (2002)
22. Li, P., et al.: Mojojojo — More ownership for multiple owners. In: FOOL (2010)
23. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL (2006)
24. Lucassen, J., et al.: Polymorphic effect systems. In: POPL (1988)
25. Meyer, B.: Systematic concurrent object-oriented programming. In: CACM (1993)
26. Moors, A., et al.: Generics of a higher kind. In: OOPSLA (2008)
27. Potanin, A., et al.: Generic ownership for generic Java. In: OOPSLA (2006)
28. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O’Reilly Media, Sebastopol (2007)
29. Singh, J., et al.: SPLASH: Stanford parallel applications for shared-memory. Technical report, Stanford Univ. (1992)
30. Smith, L., Bull, J.: A multithreaded Java grande benchmark suite. In: Third Workshop on Java for High Performance Computing (2001)
31. Smith, M.: Towards an effects system for ownership domains. ECOOP (2005)
32. Sun Microsystems, Inc. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc. (March 2008)

33. Terauchi, T., Aiken, A.: A capability calculus for concurrency and determinism. In: TOPLAS (2008)
34. Thies, W., et al.: StreamIt: A language for streaming applications. In: CC (2002)
35. Thomas, P., Weedon, R.: Object-Oriented Programming in Eiffel, 2nd edn. Addison-Wesley Longman, Amsterdam (1998)
36. Vakilian, M., et al.: Inferring method effect summaries for nested heap regions. In: ASE (2009)
37. Wadler, P.: Linear types can change the world! In: Working Conf. on Programming Concepts and Methods (1990)

Tunable Static Inference for Generic Universe Types

Werner Dietl¹, Michael D. Ernst¹, and Peter Müller²

¹ University of Washington
{wmdietl,mernst}@cs.washington.edu,
² ETH Zurich
Peter.Mueller@inf.ethz.ch

Abstract. Object ownership is useful for many applications, including program verification, thread synchronization, and memory management. However, the annotation overhead of ownership type systems hampers their widespread application. This paper addresses this issue by presenting a tunable static type inference for Generic Universe Types. In contrast to classical type systems, ownership types have no single most general typing. Our inference chooses among the legal typings via heuristics. Our inference is tunable: users can indicate a preference for certain typings by adjusting the heuristics or by supplying partial annotations for the program. We present how the constraints of Generic Universe Types can be encoded as a boolean satisfiability (SAT) problem and how a weighted Max-SAT solver finds a correct Universe typing that optimizes the weights. We implemented the static inference tool, applied our inference tool to four real-world applications, and inferred interesting ownership structures.

1 Introduction

Aliasing — multiple references to the same object — makes it hard to build complex object structures correctly and to guarantee invariants about their behavior. For example, mutation of an object through one reference can be observed through other references. This leads to problems in many areas of software engineering, including program verification, concurrent programming, and memory management.

Object ownership [10] structures the heap hierarchically to control aliasing and access between objects. Ownership type systems express properties of the heap topology, for instance whether two instances of a list may share node objects. Such information is needed to show the correctness of a coarse-grained locking strategy, where the lock of the list protects the state of all its nodes [6]. Ownership type systems also enforce encapsulation, for instance, by forcing all modifications of an object to be initiated by its owner. Such guarantees are useful to maintain invariants that relate the state of multiple objects [33]. To obtain these benefits, ownership type systems require considerable annotation overhead, which is a significant burden for software engineers.

Helping software engineers to transition from un-annotated programs to code that uses an ownership type system is crucial to facilitate the adoption of ownership type systems. Standard techniques for static type inference [12] are not applicable. First, there is no need to check for the existence of a correct typing; a flat ownership structure gives a trivial typing. Second, there is no notion of a best or most general ownership typing. In realistic implementations, there are many possible typings and corresponding ownership structures, and the preferred one depends on the intent of the programmer. Ownership inference needs to support the developer in finding desirable structures by suggesting possible structures and allowing the programmer to guide the inference.

This paper presents static inference for Generic Universe Types [14,13], a lightweight ownership type system designed to enable program verification [32]. Our static inference builds a constraint system that is solved by a SAT solver. An important virtue of our approach is that the static inference is *tunable*; the SAT solver can be provided with weights that express the preference for certain solutions. These weights can be determined by general heuristics (for instance, to prefer deep ownership for fields and general typings for method signatures), by partial annotations, through a runtime analysis, or through interaction with the programmer.

The main contributions of this paper are:

- Static Inference:** an encoding of the Generic Universe Types rules into a constraint system that can be solved efficiently by a SAT solver to find possible annotations.
- Tunable Inference:** use of heuristics and programmer interaction to indicate which among many legal solutions is preferable; this approach is implemented by use of a weighted Max-SAT solver.
- Evaluation:** an implementation of our inference scheme on top of the OpenJDK compiler, and an illustration of its effectiveness on real programs.

This paper is organized as follows. Sec. 2 gives background on Generic Universe Types. Sec. 3 overviews the inference system using examples. Sec. 4 formalizes the static inference, consisting of the core programming language, the constraint generation rules, the weighting heuristics, and the encoding as a weighted SAT problem. Sec. 5 describes our implementation and our experience with it. Finally, Sec. 6 discusses related work, and Sec. 7 concludes.

2 Background on Generic Universe Types

Generic Universe Types (GUT) [14,13] is an ownership type system that allows programmers to describe and enforce hierarchical heap topologies and optionally enforces the owner-as-modifier encapsulation discipline based on the topology. GUT is integrated into the tool suite of the Java Modeling Language (JML) [23].

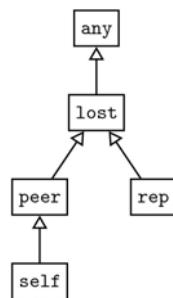
```

public class Person {
    peer Person spouse;
    rep Account savings;
    rep List<peer Person> friends;

    int assets() {
        any Account a = spouse.savings;
        return savings.balance + a.balance;
    }
}

```

(a) Example program.



(b) Ownership modifier type hierarchy.

Fig. 1. (a) A simple example of Generic Universe Types. A `Person` object owns its savings account and has the same owner as its spouse. It also owns a `List` of `Person` objects, each of which is its peer. (b) The type hierarchy of the ownership modifiers; see Sec. 2 for an explanation.

Ownership Topology. GUT organizes the heap hierarchically into contexts and restricts modifications across context boundaries. As in most other ownership systems, each object has at most one owner object. The ownership relation is acyclic. A *context* is the set of objects sharing an owner.

In GUT, a programmer expresses the ownership topology by writing one of three *ownership modifiers* on each reference type. An ownership modifier expresses ownership relative to the current receiver object `this`.

peer expresses that the referenced object is in the same context as the current object `this`. For example, in Fig. 1a, a `Person p` has the same owner as `p.spouse`.

rep expresses that the referenced object is owned by the current object. For example, in Fig. 1a, a `Person p` is the owner of `p.savings`.

any gives no static information about the relationship of the two objects.

In addition, the formalization uses two internal ownership modifiers, which are not part of the surface syntax:

lost expresses that the two objects have a relationship, but that relationship is not expressible as **peer** or **rep**. For example, in Fig. 1a, `spouse.savings` is a “nephew” of `this`; GUT cannot express this relationship, so it gives `spouse.savings` the ownership modifier **lost**.

self is used only for the current receiver object `this`.

Fig. 1b gives the type hierarchy. A **self**-modified type is a subtype of the corresponding **peer** type because **self** denotes the `this` object, which is obviously a peer of `this`. Types with **self**, **rep**, and **peer** modifiers are subtypes of the corresponding type with a **lost** modifier because **lost** conveys less ownership information. Similarly, an **any**-modified type is a supertype of all other versions.

The example in Fig. 1a also illustrates the use of ownership modifiers with generic types. Field `friends` has type `rep List<peer Person>`, which expresses

that the `List` object is owned by the `Person` object containing the field, whereas the elements stored in the list are peers of that object. Note that the ownership modifier of a type argument is interpreted relative to the client that instantiates the generic type (here, the `Person` object), not the object of the generic type.

Compound expressions: viewpoint adaptation and `lost`. The modifier of a compound expression is determined by combining the ownership modifiers of its components. For example, consider a field access `tony.spouse`, where `tony` is of type `rep Person`. This expression traverses first a `rep` reference and then a `peer` reference, so its modifier is the result of adapting `tony`'s `spouse` modifier from the viewpoint of `tony` (where it is `peer`) to the viewpoint of `this`. Here, this adaptation yields `rep` because the resulting object has the same owner as `tony`, which is `this`.

In some cases, this *viewpoint adaptation* leads to a loss of static ownership information. For example, the expression `spouse.savings` traverses first a `peer` and then a `rep` field, so the resulting object has a specific relationship to `this`, but the relationship cannot be expressed in the type system. GUT uses a special ownership modifier `lost` to express this. Two different expressions of `lost` type might stand for different unknown relationships, so it would be illegal to assign one `lost` expression to another one. GUT remains sound by prohibiting the `lost` type on the left-hand side of an assignment. This explains why GUT introduces `lost` rather than reusing `any` to stand for an unknown relationship: it would be too restrictive to forbid all assignments to left-hand-sides of type `any`.

Formally, viewpoint adaptation is a function \triangleright that takes two ownership modifiers and yields the adapted modifier. (1) `peer` \triangleright `peer` = `peer`; (2) `rep` \triangleright `peer` = `rep`; (3) `u` \triangleright `any` = `any`; (4) `self` \triangleright `u` = `u`; and (5) for all other combinations the result is `lost`. In Fig. 11a, the modifier of `spouse.savings` is `peer` \triangleright `rep` = `lost`. Since `lost Person` is a subtype of `any Person`, the expression may be assigned to variable `a`.

In addition to field accesses, viewpoint adaptation also occurs for parameter passing, result passing, and type variable bound checks.

Encapsulation. Generic Universe Types enforce that programs adhere to the heap topology described by the ownership modifiers. In addition, they optionally enforce an encapsulation scheme called the *owner-as-modifier* discipline [15]: an object `o` may be referenced by any other object, but reference chains that do not pass through `o`'s owner must not be used to modify `o`. This allows owner objects to control state changes of owned objects and thus maintain invariants. For instance, a `Person` object can enforce the invariant `savings.balance` ≥ 0 because the owner-as-modifier discipline prevents aliases to the `Account` object `savings` from modifying its `balance` field. Therefore, it is sufficient to check that each method of the `Person` object maintains the invariant.

The owner-as-modifier discipline is enforced by forbidding field updates and non-pure (side-effecting) method calls through a `lost` or `any` reference. For instance, the call `spouse.savings.withdraw(1000)` is rejected by the type system because the viewpoint-adapted modifier of the receiver, `spouse.savings`, is

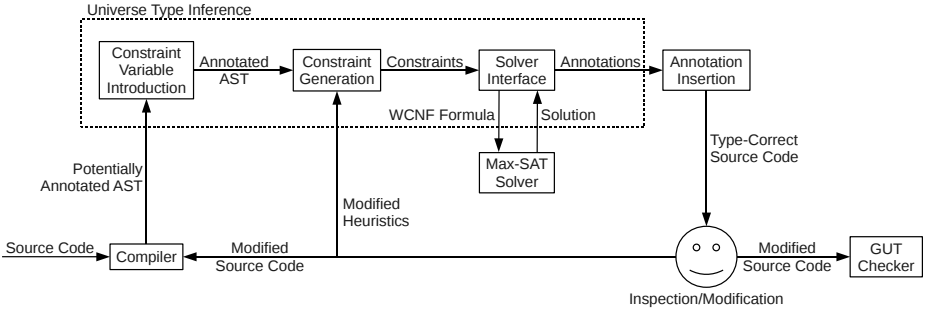


Fig. 2. Overview of the inference approach. See Sec. 3 for a detailed discussion.

`lost`. A `lost` or `any` reference can still be used for field accesses and to call pure (side-effect-free) methods. For instance, method `assets` in Fig. 1a may read the `balance` field via the `any` reference `a`.

Because the default modifier is `peer`¹, an un-annotated Java program is a legally-typed program in GUT. This typing describes a flat ownership structure — no object is owned by any other object — and so it imposes no constraints on, nor guarantees about, the program’s operation. Therefore, inference is needed to automatically produce annotations that express a deeper ownership structure.

3 Inference Approach and Example

Given a Java program as input, which may be partially annotated with ownership modifiers, the static inference determines a legal GUT typing. Fig. 3 shows an example input program and four inferred typings. Fig. 2 overviews the process. Sec. 3.1 discusses the type inference process (the dotted rectangle of Fig. 2), which is the focus of this paper. Sec. 3.2 explains how a user can iteratively use our toolset (the rest of Fig. 2).

3.1 Inference Approach

Type inference has three main steps: creating constraint variables, creating constraints over those variables, and solving the constraints to infer a typing.

The inference first creates a constraint variable for each possible occurrence of an ownership modifier or of viewpoint adaptation in the source code. Recall that GUT allows ownership modifiers for all reference types. Primitive types and type variable declarations/uses do not take ownership modifiers, and therefore no constraint variables are created for them. However, the upper bound of a type variable takes ownership modifiers. In the example in Fig. 3, a total of 14 constraint variables are introduced: α_1 – α_{11} correspond to the locations where ownership annotations may appear in the source code and α_{12} – α_{14} are constraint

¹ There are a few exceptions. For instance, subtypes of `Throwable` have the `any` modifier by default to allow the propagation of exceptions across ownership contexts.

```

class Person {
  Person1 spouse;
  Account2 savings;
  List3<Person4>12 friends;

  void marry(Person5 p) {
    spouse = p;
  }

  void befriend(Person6 p) {
    friends.add(p);
  }
}

int assets() {
  Account7 a = spouse.savings13;
  return savings.balance + a.balance;
}

void demo() {
  Person8 o1 = new Person9();
  Person10 o2 = new Person11();
  this.marry(o1);
  o1.befriend(o2)14;
}

```

Solution	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8	α_9	α_{10}	α_{11}	α_{12}	α_{13}	α_{14}
no weights	peer	peer	peer	peer	peer	peer	peer	peer	peer	peer	peer	any	peer	peer
default	any	rep	rep	any	any	any	any	rep	rep	rep	rep	any	lost	any
alternative	rep	rep	rep	peer	rep	peer	any	rep	rep	rep	rep	any	lost	rep
manual	<i>peer</i>	rep	rep	any	peer	any	any	peer	peer	rep	rep	any	lost	any

Fig. 3. An example un-annotated program and typings. Our algorithm creates constraint variables α_1 – α_{11} corresponding to locations 1–11 of type uses, and creates α_{12} – α_{14} corresponding to the viewpoint adaptations induced by the expressions at locations 12–14. The figure also shows four inference solutions that are inferred by our tool, depending on the heuristics used and whether the programmer provides a partial annotation. In the last solution, the italic *peer* modifier for α_1 was manually added to the source code before inferring the solution.

variables for viewpoint adaptation. For example, a constraint variable α_1 is introduced for the ownership modifier in the `spouse` field type, and constraint variable α_9 is introduced for the ownership modifier in the `new` expression. One generic type contains multiple constraint variables corresponding to the main and type argument modifiers; for the type of field `friends`, α_3 is introduced for the main modifier and α_4 is introduced for the type argument. Viewpoint adaptation introduces additional constraint variables, for example, α_{12} represents the result of adapting the declared upper bound of the type variable of class `List` (assumed to be `any Object`) from the point-of-view of `friends` to `this`.

The inference generates constraints over the constraint variables by traversing the AST. These constraints correspond one-to-one to the type rules of GUT [13]. Additional, weighted, breakable constraints express preferences regarding the solution, obtained by applying a heuristic. For Fig. 3, the tool generates a total of 35 constraints. For example, the assignment `spouse = p` results in a subtype constraint between constraint variables α_5 and α_1 . Constraints are also generated to connect the constraint variables involved in viewpoint adaptation. For example, the field access `spouse.savings` has to adapt the declared type of field `savings` from the point-of-view of `spouse` to `this`. To model the result of this adaptation, the additional constraint variable α_{13} was introduced.

The constraint $\alpha_1 \triangleright \alpha_2 = \alpha_{13}$ is generated to encode the dependency between the involved variables. The assignment to local variable `a` then induces a subtype constraint between α_{13} and α_7 .

The constraints are translated into a weighted SAT formula. A weighted Max-SAT solver [5] finds a solution that satisfies all of the type system constraints and the breakable constraints resulting in the maximum weight. The SAT solution is translated into a typing for the program: a concrete ownership modifier for each constraint variable in the program. Any such typing is guaranteed to be correct, as all type rules are encoded as mandatory constraints; our experiments also confirm this.

3.2 Iterative Usage

As explained in the introduction, the best typing in an ownership type system depends on the programmer’s intent and how the annotations will be used by downstream tools. Therefore, we expect the inference tool to be used iteratively.

Using the inference without heuristics results in the first solution presented in Fig. 3: all modifiers are assigned `peer`, except for α_{12} which is the result of adapting the declared `any` upper bound. This flat assignment is usually not the desired solution.

By using our tool’s built-in heuristics (see Sec. 4.3), we get the second result in Fig. 3. This heuristic prefers a deep ownership structure and broadly applicable methods. For example, this heuristic prefers α_5 and α_6 to be `any`, making the methods callable with arbitrary arguments, even though this solution requires α_1 and α_4 to be `any`, reducing the encapsulation guarantees of these fields. α_2 and α_3 are inferred to be `rep`, as they are not dependent on parameter types.

If the inferred annotations do not reflect the programmer’s design intent, the programmer can improve the result in three ways: customize the heuristics, manually add annotations, or fix defects in the source code.

(1) The programmer may customize the heuristics to encourage certain results. For instance, the programmer might select heuristics that favor general types for a library in order to make the library as widely applicable as possible. By contrast, when the whole program is available, the programmer might select heuristics that favor restrictive modifiers to facilitate subsequent use of the ownership information, for instance by a program verifier. In our example, the third solution was inferred using a heuristic that prefers `rep` as annotation for fields and has no other weights. Note how in this solution α_4 is inferred to be `peer`. A `rep` solution is not possible in this location, as it would result in α_6 to be `rep`, which would make the call `o1.befriend(o2)` impossible. On the other hand, α_5 was inferred to be `rep`; this was possible, because the only call of `marry` is on a `this` receiver. Calls of `marry` with a receiver other than `this` are impossible in this solution. This changed heuristic provides stronger encapsulation, but limits the future use of fields and methods. In Sec. 4.3 we will discuss weighting in more detail.

(2) The programmer can write ownership modifiers in the input program. The inference system always respects such annotations, even if the program text

would have permitted another solution. In our example, the fourth solution is the result of manually adding a `peer` annotation to field `spouse`, fixing constraint variable α_1 to `peer`. Using our default heuristics again, this would result in the parameter to `marry` also becoming `peer`. Also the type of `o1` needs to be inferred to be `peer` in order for `this` and `o1` to be peers in the call of `marry`.

(3) The programmer might fix defects in the source code. One variety of defect is programmer-written type annotations that are incompatible with one another or with usage by the source code. The inference system points out such errors (currently by providing no inference result; improved error messages are future work). In our example, assume the programmer manually adds a `peer` annotation to field `spouse` and a `rep` annotation to parameter `p` of method `marry`; no assignment to the remaining constraint variables could resolve the mismatch of modifiers in the assignment `spouse = p`.

A more subtle defect is one that prevents inference of the programmer's intended ownership structure, even though the system outputs some legal typing of the program. For example, suppose that a programmer designed a layered system in which lower layers do not call non-pure methods of higher layers. A violation of this property would cause the inference to produce a flatter-than-desired ownership structure. The programmer could correct the source code so that it implements the design.

Once the programmer is satisfied with the results, our tool inserts the ownership modifiers into the source code to improve the documentation of the code, to encourage that the heap topology and encapsulation are considered during program maintenance, and to make them available to downstream tools such as a program verifier. The GUT type checker can be used to ensure that the annotations remain consistent.

4 Tunable Static Inference

This section formalizes our inference approach. Sec. 4.1 presents a core calculus for a Java-like programming language, which is used by the rest of the formalism. Sec. 4.2 gives syntax-directed type inference rules: each programming language construct gives rise to a set of constraint variables and to a set of constraints over the variables. We introduce a constraint variable for each location in the source program where a concrete ownership modifier may be written and for each expression that requires viewpoint adaptation. Any solution to the constraints is a legal assignment of a concrete ownership modifier to each source location. Sec. 4.3 describes how to add additional constraints that express preferences among the possible solutions. Finally, Sec. 4.4 shows how to encode all the constraints as a weighted SAT problem, and to transform a weighted Max-SAT solver's output into a set of concrete ownership modifiers for the program.

4.1 Programming Language

Fig. 4 summarizes the syntax of the language and the naming conventions.

P	$::= \overline{Cls}$		
Cls	$::= \text{class } Cid \langle \overline{TP} \rangle \text{ extends } C \langle \overline{T} \rangle \{ \overline{fd} \ \overline{md} \}$	C	$::= Cid \mid \text{Object}$
TP	$::= X \text{ extends } N$	fd	$::= T \ f;$
md	$::= p \langle \overline{TP} \rangle T_r \ m \langle \overline{T} \ \overline{pid} \rangle \{ e \}$	p	$::= \text{pure} \mid \text{impure}$
e	$::= \text{null} \mid x \mid \text{new } N() \mid e.f \mid e_0.f := e_1 \mid$ $e_0. \langle \overline{T} \rangle m \langle \overline{e} \rangle \mid (N) \ e$	T	$::= N \mid X$
u	$::= \alpha \mid \text{peer} \mid \text{rep} \mid \text{any} \mid \text{lost} \mid \text{self}$	N	$::= u \ C \langle \overline{T} \rangle$
		x	$::= pid \mid \text{this}$
pid	parameter identifier	f	field identifier
m	method identifier	Cid	class identifier
α	constraint variable identifier	X	type variable identifier

Fig. 4. Syntax of our programming language. A sequence of A elements is denoted as \overline{A} . The surface syntax (written by the programmer) does not include ownership modifiers α , **lost**, or **self**, and allows omitting ownership modifiers. The only difference from previous formalizations of GUT [13] is the addition of constraint variables α as a placeholder for a concrete ownership modifier.

A program P consists of a sequence of class declarations; P is implicitly available in all judgments. A class declaration Cls names the class and its superclass, along with their type parameters and type arguments, respectively, and gives field and method declarations. A field declaration is a simple pair of a type and an identifier. A method declaration consists of the method purity, method type parameters if any, return type, method name, formal parameter declarations, and an expression for the method body. An expression e can be the **null** literal, a method parameter access, object creation, field read, field update, method call, or cast.

A type T is either a non-variable type N or a type variable X . A non-variable type N consists of an ownership modifier u and a possibly-parameterized class C . The definition of the ownership modifiers is the only deviation from previous formalizations of GUT. Ownership or Universe modifiers u include the *concrete ownership modifiers* **peer**, **rep**, **any**, **lost**, and **self**, as well as constraint variables α . Constraint variables α are used as placeholders for the concrete ownership modifiers that the system will infer. The surface syntax does not include α , **lost**, or **self** and allows omitting ownership modifiers; constraint variables are introduced for all omitted ownership modifiers.

4.2 Building the Constraints

This section introduces constraint variables (Sec. 4.2.1), the kinds of constraints (Sec. 4.2.2), and the syntax-directed rules that build the constraints (Sec. 4.2.3).

4.2.1 Constraint Variables

A *constraint variable* represents the ownership modifier for the occurrence of a reference type or a particular expression.

For each position where a concrete ownership modifier may occur in the solution — that is, for each use of a type — our tool introduces a constraint variable α that represents the ownership modifier for that position. Our inference will

later assign one of the concrete ownership modifier `rep`, `peer`, or `any` to each of these constraint variables. The tool also introduces a constraint variable for each expression that induces viewpoint adaptation; these will be assigned to `rep`, `peer`, `any`, or `lost`. `self` is used only as the type of the `this` literal and never inferred.

To infer the ownership modifiers for the program of Fig. 3, our tool would introduce a constraint variable for each numbered location.

If the programmer has partially annotated the program, then the generated constraints use the programmer-written modifier instead of creating a constraint variable.

4.2.2 Constraints

The inference rules (Sec. 4.2.3) create five kinds of constraints over the ownership modifiers, in particular, over the constraint variables.

Subtype ($u_1 <: u_2$): A subtype constraint enforces that u_1 will be assigned an ownership modifier that is a subtype of the ownership modifier assigned to u_2 . Subtype constraints are used for assignments and for pseudo-assignments (parameter passing, result passing, type variable bound checks).

Adaptation ($u_1 \triangleright u_2 = \alpha_3$): An adaptation constraint ensures that the viewpoint adaptation of variable u_2 from the viewpoint expressed by u_1 results in α_3 .

Equality ($u_1 = u_2$): An equality constraint ensures that two modifiers are the same. They are used to handle method overriding and type argument subtyping, which are both invariant.

Inequality ($u_1 \neq u_2$): An inequality constraint ensures that two modifiers differ. For example, the type system forbids the `lost` modifier on the left-hand side of an assignment. The type system also forbids the `any` modifier for the receiver of field updates, if the owner-as-modifier discipline is enforced.

Comparable ($u_1 <:> u_2$): A comparable constraint expresses that two ownership modifiers are not incompatible, that is, one could be a subtype of the other. These constraints are used for casts.

Fig. 6 in Sec. 4.2.3 defines helper judgments that lift these constraints from ownership modifiers to types.

4.2.3 Constraint generation

Our system takes as input a program and creates a set Σ of the kinds of constraints defined in Sec. 4.2.2. The constraints in Σ are satisfied by any correct GUT typing for the program. The constraints correspond to the type rules [13] expressed abstractly over the ownership modifiers.

Fig. 5 contains the rules for extracting constraints from a program. It defines judgments over class, field, and method declarations, as well as over expressions. Our inference is a type-based analysis [36] that runs only on valid Java programs. Therefore, our rules do not encode all Java type rules, but give only constraints for the additional checks for Generic Universe Types. To simplify the notation, the rules use helper judgments and functions that lift constraints from single

ownership modifiers to types; they are defined in Figs. 6 and explained after the discussion of the main judgments.

An environment Γ maps type variables of the enclosing class and method to their upper bounds and variables to their types. We use the notation $\Gamma(X)$ and $\Gamma(x)$ to look-up the upper bound of a type variable and the type of a variable, respectively. Helper function `env` (defined in Fig. 6) defines the environment necessary for checking class and method declarations.

We now discuss the rules of Fig. 5.

The constraints for a *class*, *field*, *method parameter*, and *method declaration* consist of the constraints for their components. The well-formedness of types is ensured using the well-formed type (OK) judgment defined in Fig. 6. For a method declaration, note that the environment is extended with the method type variables and the method parameters. Function overriding requires that, if the current method is overriding a method in a superclass, the parameter and return types are consistent. The resulting constraint set Σ_2 defines equality constraints between the types in the current method signature and a directly overridden method signature. For space reasons, we do not show the formal definition of overriding; it follows directly from the GUT formalization [13].

Finally, there are eight judgments for *expressions*, which are also mostly standard. We discuss casts immediately below. For an object creation expression the main ownership modifier has to be different from `lost` and `any` to ensure that either `peer` or `rep` is inferred, giving the new object a specific location in the ownership topology. Helper functions `fType` and `mType`, discussed below, yield the field type, respectively the method signature, after viewpoint adaptation, and additional constraints that encode the necessary adaptations of modifiers. To ensure soundness of the inferred results, `lost` has to be forbidden for all types involved in pseudo-assignments: the adapted field type, and the adapted method parameter types and method type variable bounds. These constraints ensure that modifications are only possible if the ownership is known statically. The rules for a field update and for an impure method call generate additional constraints only when the owner-as-modifier discipline is enforced: the main modifier of the receiver expression has to be different from `lost` and `any` to ensure that the owner of the modified object is statically known.

The $\Gamma \vdash N \langle : \rangle T_0 : \Sigma_c$ clause of the cast rule requires explanation. Recall that a cast is a type loophole that indicates that the program’s behavior is beyond the reasoning capabilities of the type system. If the un-annotated input program contains a cast, then the corresponding runtime check might fail at run time. Generic Universe Types also support casts: downcasts that specialize ownership information (that is, casts from `any` to `peer` or `rep`) and require a runtime check. Our inference never inserts a new cast; to do so would defeat the purpose of static ownership type checking. However, the inference is permitted to choose arbitrary² ownership modifiers at existing casts, and therefore an existing cast might fail either because of the base language check, or because of the ownership

² Actually, the choice is not arbitrary. The $\langle : \rangle$ constraint requires the two types are comparable — otherwise, the cast is guaranteed to fail.

Environment: $\Gamma = \{\overline{X} \mapsto \overline{N}; \overline{x} \mapsto \overline{T}\}$

Class declaration: $\boxed{\vdash \text{Cls} : \Sigma}$
$$\frac{\text{env}(\text{Cid}, \overline{TP}) = \Gamma \quad \Gamma \vdash \overline{fd} : \Sigma_f \quad \Gamma \vdash \overline{md} : \Sigma_m \quad \Gamma \vdash \mathbf{self} \ C \langle \overline{T} \rangle, \text{bounds}(\overline{TP}) \text{ OK} : \Sigma_t \quad \Sigma = \Sigma_f \cup \Sigma_m \cup \Sigma_t}{\vdash \mathbf{class} \ \text{Cid} \langle \overline{TP} \rangle \ \mathbf{extends} \ C \langle \overline{T} \rangle \ \{ \overline{fd} \ \overline{md} \} : \Sigma}$$

Field and method parameter declaration: $\boxed{\Gamma \vdash T \ f : \Sigma}$, $\boxed{\Gamma \vdash T \ \text{pid} : \Sigma}$

$$\frac{\Gamma \vdash T \ \text{OK} : \Sigma}{\Gamma \vdash T \ f : \Sigma} \quad \frac{\Gamma \vdash T \ \text{OK} : \Sigma}{\Gamma \vdash T \ \text{pid} : \Sigma}$$

Method declaration: $\boxed{\Gamma \vdash \text{md} : \Sigma}$
$$\frac{\text{env}(\Gamma, \overline{TP}, \overline{T \ \text{pid}}) = \Gamma' \quad \Gamma' \vdash \overline{T \ \text{pid}} : \Sigma_0 \quad \Gamma' \vdash e : T, \Sigma_1 \quad \text{overriding}(\Gamma', m) = \Sigma_2 \quad \Gamma' \vdash \text{bounds}(\overline{TP}), T_r \ \text{OK} : \Sigma_3 \quad \Gamma' \vdash T <: T_r : \Sigma_4}{\Gamma \vdash p \ \langle \overline{TP} \rangle \ T_r \ m(\overline{T \ \text{pid}}) \ \{ e \} : \bigcup_{i=0}^{i=4} \Sigma_i}$$

Expressions: $\boxed{\Gamma \vdash e : T, \Sigma}$

$$\frac{\Gamma \vdash e : T_0 : \Sigma_0 \quad \Gamma \vdash T_0 <: T : \Sigma_1}{\Gamma \vdash e : T, \Sigma_0 \cup \Sigma_1} \quad \Gamma \vdash \mathbf{null} : T, \emptyset \quad \Gamma \vdash x : \Gamma(x), \emptyset \quad \frac{\Gamma \vdash e_0 : T_0, \Sigma_0 \quad \Gamma \vdash N \ \text{OK} : \Sigma_t \quad \Gamma \vdash N <:> T_0 : \Sigma_c \quad \Sigma = \Sigma_0 \cup \Sigma_t \cup \Sigma_c}{\Gamma \vdash (N) \ e_0 : N, \Sigma}$$

$$\frac{\Gamma \vdash e_0 : N_0, \Sigma_0 \quad \text{fType}(N_0, f) = T_2, \Sigma_2 \quad \Gamma \vdash e_1 : T_1, \Sigma_1 \quad \Gamma \vdash T_1 <: T_2 : \Sigma_3 \quad \Sigma_4 = \{\mathbf{lost} \notin T_2\}}{\Gamma \vdash e_0.f := e_1 : T_2, \bigcup_{i=0}^{i=5} \Sigma_i} \quad \frac{\Gamma \vdash N \ \text{OK} : \Sigma_0 \quad \Sigma_1 = \{\text{om}(N) \neq \{\mathbf{lost}, \mathbf{any}\}\}}{\Gamma \vdash \mathbf{new} \ N() : N, \Sigma_0 \cup \Sigma_1} \quad \Sigma_5 = \boxed{\{\text{om}(N_0) \neq \{\mathbf{lost}, \mathbf{any}\}\}}$$

$$\frac{\Gamma \vdash e_0 : N_0, \Sigma_0 \quad \text{mType}(N_0, m, \overline{T}) = p \ \langle \overline{TP} \rangle \ T_r \ m(\overline{T_p \ \text{pid}}), \Sigma_2 \quad \Gamma \vdash \overline{e_a} : \overline{T_a}, \Sigma_1 \quad \Gamma \vdash \overline{T_a} <: \overline{T_p} : \Sigma_3 \quad \Sigma_4 = \{\mathbf{lost} \notin (\overline{T_p}, \text{bounds}(\overline{TP}))\}}{\Gamma \vdash \overline{T} \ \text{OK} : \Sigma_5 \quad \Gamma \vdash \overline{T} <: \text{bounds}(\overline{TP}) : \Sigma_6 \quad p = \mathbf{impure} \Rightarrow \Sigma_7 = \boxed{\{\text{om}(N_0) \neq \{\mathbf{lost}, \mathbf{any}\}\}} \quad p = \mathbf{pure} \Rightarrow \Sigma_7 = \emptyset}{\Gamma \vdash e_0.\langle \overline{T} \rangle m(\overline{e_a}) : T_r, \bigcup_{i=0}^{i=7} \Sigma_i} \quad \frac{\Gamma \vdash e : N_0, \Sigma_0 \quad \text{fType}(N_0, f) = T, \Sigma_1}{\Gamma \vdash e.f : T, \Sigma_0 \cup \Sigma_1}$$

Fig. 5. Constraint generation rules. Helper judgments and functions are defined in Figs. 6. The generated constraint set Σ encodes all constraints that need to be fulfilled to give a valid GUT program. The two framed constraints only need to be generated if the owner-as-modifier discipline should be enforced.

check. For instance, if the inferred modifier of variables x and o are `peer` and `any`, respectively, then the constraint for the expression $x = (\text{Person}) o$ infers `peer` as ownership modifier for the cast to make the assignment type-correct. Our choice is a natural extension of the base language behavior. An alternative would be for the static inference to choose modifiers in such a way as to guarantee that the runtime ownership check at each cast succeeds. This can be accomplished by simply changing “ $\langle : \rangle$ ” to “ $=$ ”. Subsumption of the expression type could then still be used to cast to a supertype, which is guaranteed to succeed.

Helper judgments and functions. Fig. 6 defines additional judgments and functions that support the main ones of Fig. 5.

Function `om` gives the main ownership modifier for a non-variable type. Function `bounds` gives the upper bound types from type parameter declarations. We compact the notation to compare one ownership modifier against a set of ownership modifiers and to ensure that an ownership modifier does not appear in a type. Function `env` defines the environment depending on the surrounding class and method declarations.

For space reasons, we omit showing how each judgment is also lifted to sequences of elements by applying the judgment to the individual elements and combining the results.

Viewpoint adaptation is lifted from single modifiers (defined in Sec. 4.2.2) to types using two judgments: (1) adapting a type from an ownership modifier and (2) adapting a type from the viewpoint of a non-variable type. A type is adapted from the viewpoint of an ownership modifier to `this`, giving an adapted type and a constraint set. There are two cases. No constraint is generated to adapt type variables X , as they do not need to be adapted. The constraints to adapt a non-variable type $u' C\langle \overline{T} \rangle$ from viewpoint u consist of the constraint for combining u with the main modifier u' , resulting in a fresh constraint variable α , and recursively adapting the type arguments. A type is adapted from the viewpoint of a non-variable type to `this`, by first adapting the type using the main modifier u and then substituting the type arguments \overline{T} for the type variables \overline{X} . Function `typeVars` gives the type variables defined by a class. The notation $T[\overline{T}/\overline{X}]$ is used to substitute type arguments \overline{T} for occurrences of type variables \overline{X} in T .

The subtyping judgment between two types determines a constraint set that has to hold in order for the two types to be subtypes. The most interesting subtyping rule is the second one, which derives a subtyping relationship from a subclassing relationship by adapting the type arguments from the superclass to the particular subtype instantiation. The subclassing relationship \sqsubseteq is the reflexive and transitive closure of the `extends` relationship of the classes; it is defined over instantiated classes $C\langle \overline{T} \rangle$, as defined in GUT [13].

Fig. 6 does not show the lifted versions of equality and comparable constraints. The equality constraint is lifted to types by simple recursion. A comparable constraint is applied to two non-variable types by first going to a common superclass and then generating a comparable constraint for the two main modifiers and equality constraints for the type arguments.

Notation:

$$\begin{aligned} \text{om}(u \ C \langle \overline{T} \rangle) &\equiv u & (u \neq \{u_1, u_2, \dots\}) &\equiv (u \neq u_1, u \neq u_2, \dots) \\ \text{bounds}(\overline{X \text{ extends } N}) &\equiv \overline{N} & (u \notin u' \ C \langle \overline{T} \rangle) &\equiv (u \neq u' \wedge u \notin \overline{T}) \end{aligned}$$

Environment definitions:

$$\begin{aligned} \text{env}(\overline{Cid}, \overline{X \text{ extends } N}) &= \{\overline{X} \mapsto \overline{N}; \text{this} \mapsto \text{self } Cid \langle \overline{X} \rangle\} \\ \text{env}(\{X_c \mapsto N_c; \overline{x} \mapsto \overline{T}\}, \overline{X \text{ extends } N}, \overline{T_p \text{ pid}}) &= \{X_c \mapsto N_c, \overline{X} \mapsto \overline{N}; \overline{x} \mapsto \overline{T}, \overline{pid} \mapsto \overline{T_p}\} \end{aligned}$$

Ownership modifier - type adaptation: $\boxed{u \triangleright T = T' : \Sigma}$

$$\frac{}{u \triangleright X = X : \emptyset} \quad \frac{\Sigma_0 = \{u \triangleright u' = \alpha\} \quad \text{fresh}(\alpha) \quad u \triangleright \overline{T} = \overline{T'} : \Sigma_1}{u \triangleright u' \ C \langle \overline{T} \rangle = \alpha \ C \langle \overline{T'} \rangle : \Sigma_0 \cup \Sigma_1}$$

Type - type adaptation: $\boxed{\Gamma \vdash N \triangleright T = T' : \Sigma}$

$$\frac{u \triangleright T = T_1 : \Sigma \quad T_1 [\overline{T}/\overline{X}] = T' \quad \text{typeVars}(C) = \overline{X}}{u \ C \langle \overline{T} \rangle \triangleright T = T' : \Sigma}$$

Subtyping: $\boxed{\Gamma \vdash T <: T' : \Sigma}$

$$\frac{\Sigma = \{u <: u', \overline{T} = \overline{T'}\}}{\Gamma \vdash u \ C \langle \overline{T} \rangle <: u' \ C \langle \overline{T'} \rangle : \Sigma} \quad \frac{C \langle \overline{X} \rangle \sqsubseteq C' \langle \overline{T}_1 \rangle \quad u \ C \langle \overline{T} \rangle \triangleright \overline{T}_1 = \overline{T'}, \Sigma}{\Gamma \vdash u \ C \langle \overline{T} \rangle <: u \ C' \langle \overline{T'} \rangle : \Sigma}$$

$$\frac{}{\Gamma \vdash X <: X : \emptyset} \quad \frac{}{\Gamma \vdash X <: \Gamma(X) : \emptyset} \quad \frac{\Gamma \vdash T <: T_1 : \Sigma_1 \quad \Gamma \vdash T_1 <: T' : \Sigma_2}{\Gamma \vdash T <: T' : \Sigma_1 \cup \Sigma_2}$$

Well-formed type: $\boxed{\Gamma \vdash T \text{ OK} : \Sigma}$

$$\frac{\Gamma \vdash \overline{T} \text{ OK} : \Sigma_0 \quad \text{typeBounds}(u \ C \langle \overline{T} \rangle) = \overline{T'}, \Sigma_1 \quad \Gamma \vdash \overline{T} <: \overline{T'} : \Sigma_2}{\Gamma \vdash u \ C \langle \overline{T} \rangle \text{ OK} : \bigcup_{i=0}^{i=2} \Sigma_i} \quad \frac{X \in \Gamma}{\Gamma \vdash X \text{ OK} : \emptyset}$$

Adaptation of a field type: $\boxed{\text{fType}(N, f) = T, \Sigma}$

$$\text{fType}(u \ C \langle \overline{T} \rangle, f) = T', \Sigma \quad \text{where} \quad \text{fType}(C, f) = T \quad u \ C \langle \overline{T} \rangle \triangleright T = T' : \Sigma$$

Adaptation of a method signature: $\boxed{\text{mType}(N, m, \overline{T}) = p \ \overline{T_p} \ T_r \ m(\overline{T_p \ \text{pid}}), \Sigma}$

$$\begin{aligned} \text{mType}(N, m, \overline{T'}) &= p \ \overline{X \text{ extends } N'} \ T'_r \ m(\overline{T'_p \ \text{pid}}), \Sigma_b \cup \Sigma_r \cup \Sigma_p \\ \text{where} \quad N &= u \ C \langle \overline{T} \rangle \\ \text{mType}(C, m) &= p \ \overline{X \text{ extends } N_b} \ T_r \ m(\overline{T_p \ \text{pid}}) \\ N \triangleright \overline{N_b} = \overline{N_0} : \Sigma_b & \quad N \triangleright T_r = T_{r0} : \Sigma_r \quad N \triangleright \overline{T_p} = \overline{T_{p0}} : \Sigma_p \\ \overline{N_0}[\overline{T'}/\overline{X}] = \overline{N'} & \quad T_{r0}[\overline{T'}/\overline{X}] = T'_r \quad \overline{T_{p0}}[\overline{T'}/\overline{X}] = \overline{T'_p} \end{aligned}$$

Adaptation of type bounds: $\boxed{\text{typeBounds}(N) = \overline{N'}, \Sigma}$

$$\text{typeBounds}(u \ C \langle \overline{T} \rangle) = \overline{N'}, \Sigma \quad \text{where} \quad \text{class } C \langle \overline{X \text{ extends } N} \rangle \dots \in P \quad u \ C \langle \overline{T} \rangle \triangleright \overline{N} = \overline{N'}, \Sigma$$

Look-up of class type variables: $\boxed{\text{typeVars}(N) = \overline{X}}$

$$\text{typeVars}(u \ C \langle \overline{T} \rangle) = \overline{X} \quad \text{where} \quad \text{class } C \langle \overline{X \text{ extends } N} \rangle \dots \in P$$

Fig. 6. Helper judgments and functions for the constraint generation rules of Fig. 5

The well-formed type (OK) judgment defines when a type T is well-formed in an environment Γ giving the constraints Σ . We omit judgments for well-formedness of environments, which are basically just well-formedness for all involved types.

The overloaded helper functions `fType`, `mType`, and `typeBounds` are defined as follows. Function `fType(C, f)` yields the declared field type of field f in class C or a superclass of C . It yields only a type, but no constraints. The overloaded function `fType(N, f)` (taking a non-variable type rather than a class as first argument) determines the type of field f adapted from viewpoint N to `this`. It results in an adapted field type and constraints on the constraint variables of the viewpoint and the constraint variables for the declared type.

Function `mType(C, m)` yields the declared method signature of method m in class C or a superclass of C . The overloaded function `mType(N, m, \overline{T})` determines the method signature of method m adapted from viewpoint N to `this` and substituting method type arguments for their type variables. It results in an adapted method signature and constraints on the constraint variables of the viewpoint and the constraint variables for the declared parameter, return, and type variable bound types, respectively.

Function `typeBounds($u C \langle \overline{T} \rangle$)` yields the upper bounds of the type variables of class C adapted from the non-variable type $u C \langle \overline{T} \rangle$ to `this` and a set of constraints.

4.3 Heuristic Choice of a Solution

For a given set of constraints, the solver may return any satisfying assignment. For completely un-annotated programs, these solutions include the trivial one that assigns `peer` to all variables. It is typically not the desired solution because it corresponds to a completely flat ownership structure.

When choosing among many possibilities to assign ownership modifiers, a human programmer is influenced by a variety of design considerations.

- A deeper ownership structure gives better encapsulation, so it is generally preferable, but it limits sharing.
- The types in method signatures influence what clients may call the method, so it is preferable for method parameters to have the `any` modifier.
- Other heuristics are possible; for instance, a verification tool based on ownership, such as Spec# [24], has different needs for invariants and pre-/post-conditions.

To reflect these design considerations, we attach weights to some constraints. All constraints of the type system are mandatory. For each constraint variable, we use the position of the variable in the AST to encode a preference for a particular solution by adding an additional breakable, weighted equality constraint.

For variables α that appear in ... :

- ... field types, the weight for $\alpha = \mathbf{rep}$ is 80.
- ... parameter types, the weight for $\alpha = \mathbf{any}$ is 150.
- ... return types, the weight for $\alpha = \mathbf{rep}$ is 30.
- ... class and method type variable bounds, the weight for $\alpha = \mathbf{any}$ is 200.

This pre-defined heuristic prefers solutions with deep ownership structures and generally applicable methods. A user may adapt these weights, either globally or for individual variables. In the example from Fig. 3, the “alternative” solution was generated using 100 as weight for `rep` for variables that appear in field types and using no other weights. As discussed previously, this alternative weighting results in stronger encapsulation at the cost of the applicability of the methods. In the rest of this paper we will use the pre-defined heuristics; examining the effect of alternative weights is interesting future work.

Weights can also be used to handle other guidance, from a user or tool, more flexibly. Suppose that a user has partially annotated a program. The annotations are encoded as mandatory equality constraints. If the partial annotations lead to unsatisfiable constraints, the tool could—after consulting the developer—convert them from mandatory into breakable constraints. This would give the inference tool the flexibility to override annotations when necessary. The programmer would then inspect what annotations needed to be changed.

4.4 Encoding for a SAT Solver

Once the constraint system Σ is generated, it needs to be solved. We encode Σ as a weighted Max-SAT problem and use an existing solver [5] for three reasons. First, Generic Universe Types allow only a fixed number of ownership modifiers; thus, constraints can easily be encoded as boolean formulas. Second, the weights allow us to encode heuristics that direct the SAT solver to produce good solutions. Third, reusing a solver allows us to benefit from all the optimizations that went into existing solvers.

This section explains how to encode the constraints Σ as boolean formulas. These formulas are then converted to conjunctive normal form, which is the input format of the SAT solver. The SAT solver either returns an assignment of booleans that satisfies the formula or notifies the user that the formula is unsatisfiable. The assignment of booleans corresponds to ownership modifiers for the variables that satisfy all constraints.

We finally turn all the formulas into the CNF format used by the Max-SAT evaluation benchmarks [27]. This format is supported by different SAT solvers, and our implementation supports changing the solver.

Encoding of constraint variables. Four boolean variables β_i^{peer} , β_i^{rep} , β_i^{any} , and β_i^{lost} represent each ownership variable α_i from the constraints. The encoding expresses that exactly one of these four booleans is assigned true:

$$(\beta^{peer} \vee \beta^{rep} \vee \beta^{any} \vee \beta^{lost}) \wedge \neg(\beta^{peer} \wedge \beta^{rep}) \wedge \neg(\beta^{peer} \wedge \beta^{any}) \wedge \neg(\beta^{peer} \wedge \beta^{lost}) \wedge \neg(\beta^{rep} \wedge \beta^{any}) \wedge \neg(\beta^{rep} \wedge \beta^{lost}) \wedge \neg(\beta^{lost} \wedge \beta^{any})$$

For every variable that will be inserted into the program, `lost` is forbidden and the encoding of the variable is accordingly simplified.

An alternative encoding would use only two booleans to encode the four possibilities. Such an encoding would have fewer variables, but more complicated clauses to encode constraints. Our encoding can be solved more efficiently [17].

Constraint	Encoding
$\alpha_1 <: \alpha_2$	$(\beta_1^{any} \Rightarrow \beta_2^{any}) \wedge (\beta_2^{peer} \Rightarrow \beta_1^{peer}) \wedge$ $(\beta_2^{rep} \Rightarrow \beta_1^{rep}) \wedge (\beta_1^{lost} \Rightarrow (\beta_2^{lost} \vee \beta_2^{any}))$
$\alpha_1 \triangleright \alpha_2 = \alpha_3$	$(\beta_1^{peer} \wedge \beta_2^{peer} \Rightarrow \beta_3^{peer}) \wedge (\beta_1^{rep} \wedge \beta_2^{peer} \Rightarrow \beta_3^{rep}) \wedge$ $(\beta_2^{any} \Rightarrow \beta_3^{any}) \wedge (\beta_2^{lost} \Rightarrow \beta_3^{lost}) \wedge (\beta_1^{any} \wedge \neg \beta_2^{any} \Rightarrow \beta_3^{lost}) \wedge$ $(\beta_1^{lost} \wedge \neg \beta_2^{any} \Rightarrow \beta_3^{lost}) \wedge (\beta_2^{rep} \Rightarrow \beta_3^{lost})$
$\alpha_1 = \alpha_2$	$(\beta_1^{peer} \Rightarrow \beta_2^{peer}) \wedge (\beta_1^{rep} \Rightarrow \beta_2^{rep}) \wedge$ $(\beta_1^{lost} \Rightarrow \beta_2^{lost}) \wedge (\beta_1^{any} \Rightarrow \beta_2^{any})$
$\alpha_1 \neq \alpha_2$	$(\beta_1^{peer} \Rightarrow \neg \beta_2^{peer}) \wedge (\beta_1^{rep} \Rightarrow \neg \beta_2^{rep}) \wedge$ $(\beta_1^{lost} \Rightarrow \neg \beta_2^{lost}) \wedge (\beta_1^{any} \Rightarrow \neg \beta_2^{any})$
$\alpha_1 <:\triangleright \alpha_2$	$(\beta_1^{peer} \Rightarrow \neg \beta_2^{rep}) \wedge (\beta_1^{rep} \Rightarrow \neg \beta_2^{peer})$

Fig. 7. For each kind of constraint (see Sec. 4.2.2), the formula that encodes it. Each constraint variable α_i is encoded by four boolean variables β_i^{rep} , β_i^{peer} , β_i^{any} , and β_i^{lost} .

Encoding of constraints. Fig. 7 defines the encoding of the constraints from the constraint set Σ into formulas over the boolean variables and follows the definitions given in Sec. 2.

We use simpler encodings when the constraint is between a variable and a concrete ownership modifier. For example, the equality constraint $\alpha_i = \mathbf{peer}$ is encoded by the formula β_i^{peer} .

Encoding of weights. We use the weighting feature of a weighted Max-SAT solver to encode the weights of Sec. 4.3.

For each mandatory constraint we use the maximum weight, which the SAT solver treats as infinity; this enforces that all the type rules are fulfilled. For each breakable constraint we use the determined weight.

5 Implementation and Experience

This section describes the implementation (Sec. 5.1), our experience with it (Sec. 5.2), and possible future work (Sec. 5.3).

The implementation, experimental setup, and results are publicly available³.

5.1 Implementation

The static inference is implemented on top of the Checker Framework [37], which is a pluggable type checking framework built on top of the JSR 308 branch of the OpenJDK compiler [16]. By building our inference tool on the OpenJDK compiler, the tool supports full Java. The implementation of many language features is significantly simplified by the Checker Framework, which provides an abstraction of a basic type checker. The annotations inferred by the tool are stored in the input format of the Annotation File Utilities (AFU), which can

³ <http://www.cs.washington.edu/homes/wmdietl/inference/>

Benchmark	SLOC	Constraint Size				CNF Size			Timing (seconds)			
		vars		constraints		vars		clauses	topol.		encap.	
				topol.	encap.			topol.	encap.	gen	solve	gen
1. zip	2611	455	2411	2949	4656	13639	14063	4.5	1.1	4.5	1.1	
2. javad	1846	364	2571	3113	4988	14989	15333	3.5	1.0	3.6	1.0	
3. jdepend	2460	824	4868	6024	9752	28110	29176	5.1	1.4	5.8	1.5	
4. classycle	4658	1548	8726	10242	17756	53062	54380	6.0	1.8	6.2	2.0	

Fig. 8. Size and timing results. SLOC gives the number of non-blank, non-comment lines as determined by the `sloc` tool. The constraint size columns give the number of constraint variables and constraints in the program. The CNF size gives the number of boolean variables and clauses in the CNF encoding. Finally, the timing columns give the time for generating (`gen`) and solving (`solve`) the constraints. We executed each run three times and report the median. The number of constraints and clauses and the timing is further sub-divided into whether annotations for only the topology or also for enforcing the encapsulation discipline should be inferred. This choice does not affect the number of constraint variables or boolean variables.

automatically insert the inferred annotations into the source code. The AFU format also facilitates the comparison of multiple runs of the inference tool. We separately implemented a Generic Universe Types checker that handles Java programs with GUT annotations.

The inference implementation consists of around 4400 non-comment, non-blank lines of Scala code. The biggest development effort was spent on introducing unique constraint variables and mapping them to AFU output positions. Constraint generation reuses a lot of the existing Checker Framework infrastructure.

Our tool is modular and only generates constraints for the part of the program that is supplied as input. For the remainder of the program, in particular for the JDK libraries, the tool currently uses the default modifier `peer`. The Checker Framework supports a library annotation mechanism and we plan to use this feature to provide a version of the JDK that is annotated with Generic Universe Types.

5.2 Experience

We applied the tool to four real-world, open source tools developed by external developers. Fig. 8 presents size and timing information and Fig. 9 presents statistics of the inferred annotations.

The four subjects are: (1) OpenJDK’s implementation of the `zip` and `gzip` compression algorithms, taken from OpenJDK 7 build 138, (2) `javad`⁴, a Java class file disassembler, (3) `JDepend`⁵, a quality metrics tool, and (4) `Classycle`⁶, a Java class dependency analyzer.

⁴ <http://www.bearcave.com/software/java/javad/>, downloaded in December 2010.

⁵ <http://www.clarkware.com/software/JDepend.html>, version 2.9.1.

⁶ <http://classycle.sourceforge.net/>, version 1.3.3.

Benchmark	Topology					Encapsulation				
	peer	rep	any	% rep	% any	peer	rep	any	% rep	% any
1. zip	306	81	68	18%	15%	322	93	40	20%	9%
2. javad	185	87	92	24%	25%	279	55	30	15%	8%
3. jdepend	529	175	120	21%	15%	600	160	64	19%	8%
4. classycle	1132	193	223	13%	14%	1165	188	195	12%	13%

Fig. 9. Number of inferred annotations, separated into inferring only the topology and also inferring the owner-as-modifier encapsulation discipline

For each subject, we inferred a solution for the ownership topology and a solution that also enforces the owner-as-modifier encapsulation discipline. For this we manually added around 300 purity annotations to the four projects; in the future we plan to integrate an automated purity analysis.

We evaluated three qualities of our tool implementation:

1. correctness of the inferred annotations,
2. usefulness of the inferred annotations, and
3. scalability with respect to performance.

Correctness. We inserted the inferred annotations into the source code and ran the GUT type checker on these programs. In each case, the type checker verified the correctness of the inference results.

The inference and type checker are independent implementations which are separately implemented on top of the Checker Framework. Each is based on the proved formalization of GUT [13].

The most notable limitation is related to raw types and local inner classes. Our tool soundly infers an ownership modifier for the missing type arguments, but the external annotation tool we use—the Annotation File Utilities—does not support adding new type arguments to a raw type; code that uses raw types might not compile. We manually added such annotations in our case studies. Additionally, local inner classes are not correctly identified by the AFU annotation tool and fail to insert; we did not encounter this problem in our case studies.

Usefulness. We manually examined the inference results and believe they accurately reflect the ownership properties of the original programs. The relatively large number of `peer` annotations in Fig. 9, indicating a flat inferred ownership structure, is expected. We ran the inference tools on un-modified programs and did not attempt to improve the structure of the programs. The programs were probably written with an intuitive sense of the desired ownership, but with no tools to help the programmer achieve that goal. Another limitation that causes a flat ownership structure is using the `peer` default modifier for libraries. Method parameters and upper bounds of type variables that were defaulted to `peer` force structures to be flatter than desired. It will be interesting future work to allow changing both the structure of the programs and improving the library annotations. Considering this, 10–20% inferred `rep` annotations is promising.

The number of **any** annotations consistently decreases when enforcing the owner-as-modifier discipline. It is interesting to observe that the number of **rep** references increased or decreased, depending on the application. Our explanation for this is that modifications that can be performed with an **any** receiver without enforcing an encapsulation discipline need to use a **peer** or **rep** receiver when an encapsulation discipline is enforced.

Performance. Parsing the Java files took the bulk of the inference time. Encoding the constraints into CNF, waiting for the SAT solver, and decoding the results used only around a quarter of the total time.

To experiment with scalability we applied the inference tool to JabRef⁷, a bibliography management tool consisting of around 74000 SLOC. The inference generated 24402 variables and 248858 constraints, which were then translated to 521152 boolean variables and 1606319 CNF clauses. Generation of the constraint system took a total of 41 seconds and solving the system took a total of 66 seconds, of which 42 seconds were spent in the SAT solver. Unfortunately, the Annotation File Utilities crash when inserting the annotations, so we cannot use the GUT type checker to verify the results.

The used hardware was a desktop machine with two CPUs, each a 4 core Intel Xeon E5405 CPU at 2.00 GHz running Fedora 13 Linux 32 bit and using OpenJDK 7 build 138. The total main memory available is 8 GB, but the Java heap space is limited to 1 GB. The maximum observed pre-GC memory consumption during constraint generation for our four case studies was around 160MB; when run on JabRef, the maximum pre-GC consumption was 510 MB. All our software is single threaded.

5.3 Future Work

Usability. We have shown initial results that our tool scales and produces correct results, but this is not enough: it must also be usable by and useful to real programmers. We plan to perform case studies and experiments to evaluate the quality of the inferred typings, to investigate what ownership structures occur in real programs, and to see how our tool is used in practice.

At the moment, the tool provides no information about an unsatisfied constraint system. We plan to exploit the ability of Max-SAT solvers to return a partially-fulfilling assignment, to direct the programmer towards conflicts in the system.

Other ownership systems. We expect that our inference approach can be adapted to other ownership type systems.

First, we plan to extend our inference to support ownership transfer [11,34], which requires inference of the uniqueness of variables and coping with dynamic changes of ownership information.

Second, we plan to investigate how our approach can be adapted to ownership-parametric type systems [3,10,39]. We are confident that by combining static

⁷ <http://jabref.sourceforge.net>, version 2.6.

and runtime inference, we can effectively determine the minimum number of ownership parameters required to type a class.

Third, we plan to explore how we can infer ownership annotations for more complex topologies such as ownership domains [2] or multiple ownership [8].

Performance. Our inference tool seems to be fast enough for its expected use case. We have made no attempt to optimize performance, so there are many opportunities to speed up the tool, if performance becomes a problem.

Integration into an IDE would give access to the internal AST. This would cut the cost of the AST generation and allow for immediate interaction with the developer.

The SAT solver is invoked as a separate process and the input CNF is written to a file and the output from the solver needs to be parsed and interpreted. The advantage of this design is that we can use an arbitrary Max-SAT solver that supports the Max-SAT evaluation format [27]. We currently use the Sat4j solver [5] and plan to experiment with tighter integration, cutting out the file generation and parsing overheads.

6 Related Work

We discuss related work on ownership inference and on other inference.

SafeJava [67] provides intra-procedural ownership type inference for local variables to reduce the annotation overhead. Agarwal and Stoller [1] describe a runtime technique that infers further annotations. In contrast, we provide a static analysis that infers all necessary annotations in a program.

AliasJava [3] combines ownership and aliasing. It uses a constraint system to infer alias annotations. A key difference to our work is that the inference for AliasJava needs to introduce ownership parameters for classes, whereas GUT expresses ownership via ownership modifiers. The inference for AliasJava potentially results in classes with many more ownership parameters than what would have been used by a programmer. The system does not support partial annotations to fix the number of ownership parameters. In contrast, GUT allows one to associate ownership modifiers with type arguments for existing type parameters, but the inference never needs to introduce additional type parameters.

The box model [38] separates the program into module interfaces and implementations. Ownership annotations are still required for the module interface, but are automatically inferred for the implementations.

Pedigree types [25] present an intricate ownership type system similar to Universe types with polymorphic type inference for annotations. It builds a constraint system that is reduced to a set of linear equations. The inference does not help with finding good ownership structures, but only helps propagate existing annotations. We believe that our approach to type inference is easier to understand and better supports the programmer in finding the desired ownership structure. Our approach also handles generic types.

Milanova [28] presented a static inference of ownership annotations, then together with Vitek extended the work to the owner-as-dominator system [29,30]. Their tool constructs a static approximation to the object graph via an alias analysis and then computes dominators to obtain candidates for owners. Milanova et al.’s work differs from ours in five major aspects. (1) They do not use one of the extant ownership type systems, nor do they formally define the meaning of the inferred annotations. This makes it difficult to compare the precision of the tools. Moreover, the correctness of the inferred annotations can only be checked manually. Our work is based on the formalization of GUT, which has been proved sound, and we used the GUT type checker to confirm the correctness of the inferred annotations. (2) Their tool infers only annotations for field declarations and `new` expressions, whereas we infer all annotations required by Generic Universe Types. For example, for the `zip/gzip` programs, their tool outputs 81 annotations whereas ours outputs 455. (3) Milanova et al. use a whole-program pointer analysis, whereas our approach is modular and thus applicable to libraries and single classes. (4) Even though Milanova et al.’s analysis is asymptotically faster ($O(n^2)$ versus the NP-completeness of SAT), our tool seems to outperform theirs in practice. For example, their tool took 27 and 28 seconds to analyze the `gzip` and `zip` programs, respectively, whereas ours took only 5.6 seconds for both programs together (for comparison, their experiments were performed on a MacBook Pro with unspecified CPU). (5) Milanova et al.’s analysis does not handle generics, whereas ours does.

Ma and Foster [26] present a static analysis that combines an intraprocedural points-to analysis and an interprocedural predicate inference to infer uniqueness and ownership properties. Their system uses a strict definition of ownership and found less than 2% of parameters to be owned; it also does not map the results to a type system.

Kacheck/J [20] infers package-level encapsulation properties. The system extracts a set of boolean constraints from a bytecode program. These constraints encode that a class is not confined (that is, its objects may be accessed outside the package that contains the class), that a method is not anonymous (that is, it potentially assigns `this` to a non-confined type), and implications between these properties. The constraint system is a set of ground Horn clauses, which is solved in linear time. The solution indicates which classes are confined. Kacheck/J and our system share the goal of inferring encapsulation properties via constraint solving, but differ in three important aspects. (1) For confined types, there is a best solution, namely the one with the largest set of confined types, whereas our system uses weights and a Max-SAT solver to compute desirable solutions. (2) Confined types support a non-hierarchical topology of static contexts, whereas GUT offers hierarchies of contexts that can be created dynamically. The enforcement of an encapsulation policy is optional in our system. (3) Confined types do not support generics, but our inference does.

Moelius and Souter’s static analysis for ownership types resulted in a large number of ownership parameters [31].

Baker [4] observes that Hindley-Milner type inference uses distinct datatype nodes to represent disjoint runtime values. Therefore, the information computed by this inference can be used to infer aliasing information. O’Callahan and Jackson’s Lackwit tool [35] uses this idea to compute aliasing information for C programs. Lackwit associates a tag with each type constructor in a program and then uses Hindley-Milner type inference to compute equalities between these tags. Variables whose types have different tags cannot be aliases. The alias information computed by Lackwit is useful for various software engineering tasks, but not sufficient to infer ownership. In particular, Lackwit cannot distinguish several instances of the same data structure, for instance, to infer whether the nodes of two list instances may be shared. Guo et al. [21] show how to perform a similar analysis dynamically, increasing precision.

General type qualifier inference [9,19] infers any solution that satisfies all constraints, which is not useful for ownership types, where a trivial solution always exists.

Transitioning from a non-generic to a generic program [22] also deals with an under-constrained type inference problem that uses heuristics to determine a good solution.

The system whose implementation is most similar to ours is a type inference system against races [18]. It builds a constraint system, uses a SAT solver to find solutions, and exploits a Max-SAT encoding to produce good error reports, in cases where the constraint system is unsatisfiable. However, they are not concerned with finding an optimal structure for their system, since any valid locking strategy is acceptable. We use the weighting mechanism to find a desirable ownership structure among satisfiable solutions.

7 Conclusion

We presented a novel approach to static ownership inference. To the best of our knowledge, each of the following points is unique to our system. (1) Our system accommodates preferences among multiple legal typings; the preferences can come from sources such as heuristics and partial annotations. (2) It uses a Max-SAT solver to encode programmer preferences and produce a desirable inference result. (3) Our system infers ownership types for an existing, formally-defined type system that supports generic types. (4) It infers complete ownership type annotations for realistic programs. (5) The system supports either only inferring an ownership topology, or also enforcing the owner-as-modifier encapsulation discipline. (6) Its results are both correct, as verified by a type checker, and desirable, as verified by manual inspection.

Acknowledgments. We thank the reviewers for their extensive feedback. Werner Dietl was supported in part by a fellowship from the Swiss National Science Foundation (SNSF). This work was also supported by NSF grant CNS-0855252.

References

1. Agarwal, R., Stoller, S.D.: Type Inference for Parameterized Race-Free Java. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 149–160. Springer, Heidelberg (2004)
2. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
3. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, pp. 311–330 (2002)
4. Baker, H.G.: Unify and conquer (garbage, updating, aliasing) in functional languages. In: LISP and functional programming (LFP), pp. 218–226 (1990)
5. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation* 7, 59–64 (2010), <http://www.sat4j.org/>
6. Boyapati, C.: SafeJava: A Unified Type System for Safe Programming. PhD thesis, MIT (2004)
7. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. In: OOPSLA, pp. 211–230 (2002)
8. Cameron, N., Drossopoulou, S., Noble, J., Smith, M.: Multiple ownership. In: OOPSLA, pp. 441–460 (2007)
9. Chin, B., Markstrum, S., Millstein, T., Palsberg, J.: Inference of user-defined type qualifiers and qualifier rules. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 264–278. Springer, Heidelberg (2006)
10. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA (1998)
11. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003)
12. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL, pp. 207–212 (1982)
13. Dietl, W.: Universe Types: Topology, Encapsulation, Genericity, and Tools. PhD thesis, Department of Computer Science, ETH Zurich (2009)
14. Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 28–53. Springer, Heidelberg (2007)
15. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)* 4(8), 5–32 (2005)
16. Ernst, M.D.: Type annotations specification (JSR 308) (September 12, 2008), <http://types.cs.washington.edu/jsr308/>
17. Ernst, M.D., Millstein, T.D., Weld, D.S.: Automatic SAT-compilation of planning problems. In: IJCAI, pp. 1169–1176 (1997)
18. Flanagan, C., Freund, S.N.: Type inference against races. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 116–132. Springer, Heidelberg (2004)
19. Greenfieldboyce, D., Foster, J.S.: Type qualifier inference for Java. In: OOPSLA, pp. 321–336 (2007)
20. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: OOPSLA, pp. 241–253 (2001)
21. Guo, P.J., Perkins, J.H., McCamant, S., Ernst, M.D.: Dynamic inference of abstract types. In: ISSTA, pp. 255–265 (2006)
22. Kiezun, A., Ernst, M.D., Tip, F., Fuhrer, R.M.: Refactoring for parameterizing Java classes. In: ICSE, pp. 437–446 (2007)

23. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML reference manual (2008), <http://www.jmlspecs.org/>
24. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
25. Liu, Y.D., Smith, S.: Pedigree types. In: IWACO (2008)
26. Ma, K., Foster, J.S.: Inferring aliasing and encapsulation properties for Java. In: OOPSLA, pp. 423–440 (2007)
27. Max-SAT evaluation input and output format (February 2010), <http://www.maxsat.udl.cat/10/requirements/>
28. Milanova, A.: Static inference of Universe types. In: IWACO (2008)
29. Milanova, A., Liu, Y.: Practical static ownership inference. Technical Report RPI/DCS-09-04, Rensselaer Polytechnic Institute (March 2010)
30. Milanova, A., Vitek, J.: Static dominance inference. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 211–227. Springer, Heidelberg (2011)
31. Moelius, S.E., Souter, A.L.: An object ownership inference algorithm and its application. In: Mid-Atlantic Student Workshop on Programming Languages and Systems (2004)
32. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
33. Müller, P., Poetzsch-Heffter, A., Leavens, G.T.: Modular invariants for layered object structures. *Science of Computer Programming* 62, 253–286 (2006)
34. Müller, P., Rudich, A.: Ownership transfer in Universe Types. In: OOPSLA, pp. 461–478 (2007)
35. O’Callahan, R., Jackson, D.: Lackwit: A program understanding tool based on type inference. In: ICSE, pp. 338–348 (1997)
36. Palsberg, J.: Type-based analysis and applications. In: PASTE, pp. 20–27 (2001)
37. Papi, M.M., Ali, M., Correa Jr., T.L., Perkins, J.H., Ernst, M.D.: Practical plug-gable types for Java. In: ISSTA, pp. 201–212 (2008)
38. Poetzsch-Heffter, A., Geilmann, K., Schäfer, J.: Inferring ownership types for encapsulated object-oriented program components. In: Reps, T., Sagiv, M., Bauer, J. (eds.) *Program Analysis and Compilation, Theory and Practice*. LNCS, vol. 4444, pp. 120–144. Springer, Heidelberg (2007)
39. Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic Java. In: OOPSLA, pp. 311–324 (2006)

Verifying Multi-object Invariants with Relationships

Stephanie Balzer and Thomas R. Gross

ETH Zurich

Abstract. Relationships capture the interplay between classes in object-oriented programs, and various extensions of object-oriented programming languages allow the programmer to explicitly express relationships. This paper discusses how relationships facilitate the verification of multi-object invariants. We develop a visible states verification technique for Rumer, a relationship-based programming language, and demonstrate our technique on the Composite pattern. The verification technique leverages the “Matryoshka Principle” embodied in the Rumer language: relationships impose a stratification of classes and relationships (with corresponding restrictions on writes to fields, the expression of invariants, and method invocations). The Matryoshka Principle guarantees the absence of transitive call-backs and restores a visible states semantics for multi-object invariants. As a consequence, the modular verification of multi-object invariants is possible.

1 Introduction

Invariants provide a foundation for verifying programs [1], and various object-oriented programming and specification languages [2–4] have adopted invariants for objects. An *object invariant* captures the properties of an object that the object exhibits in its consistent states. Object invariants are central to a wealth of object-oriented verification techniques [5–12]. A key issue for any practical verification technique for an object-oriented language is the ability to modularly verify a program so that modules (i.e., classes) can be verified independently from each other.

Modular verification is straightforward as long as an object invariant constrains only the state of the current object and provided that an object’s fields can be written to only by the object’s own methods. Unfortunately, single-object invariants rarely express the constraints of real-world software, which typically asks for multi-object invariants. A *multi-object invariant* relates several objects and constrains not only the state of the current object but also the state(s) of the object(s) it refers to. The reasoning about a multi-object invariant, however, is no longer modular. For instance, if there are aliases to the referenced objects, the referenced objects may be altered in ways compromising the invariant.

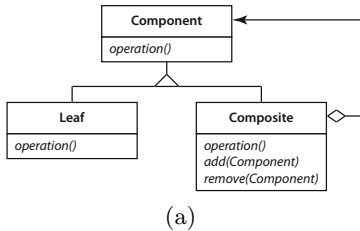
Multi-object invariants compromise also the adoption of a visible states semantics for invariants. A *visible states verification technique* [9] requires an object to meet its invariant in the initial and final states of method executions

(i.e., the visible states) but allows an object to temporarily break its invariant during the execution of a method. A visible states semantics for invariants facilitates data type induction [1, 13, 14] as a proof technique: each method may assume that the invariant holds in the method’s initial state, provided that each method ensures that the invariant holds in the method’s final state. To facilitate data type induction, a verification technique needs to guard re-entrant method invocations (*call-backs*). A call-back occurs if a method, executing on an object o , invokes a method $n()$ either directly or transitively (by further method invocations) on the original object o . Since $n()$ is invoked in a state when o ’s invariant may be temporarily broken, $n()$ should not be allowed to assume o ’s invariant in its initial state. Data type induction can be restored by imposing additional obligations on the caller of a method. Namely, a calling method $m()$ is required to re-establish the invariants of those objects O that are vulnerable to $m()$ ’s execution, provided that the objects O are possible receivers of the direct or transitive method invocation $n()$. This obligation can be easily implemented for single-object invariants by requiring a method to re-establish its receiver’s invariant before invoking a method. However, imposing the same obligation for multi-object invariants is (i) generally infeasible since the transitive receivers of method invocations are statically unknown, and also (ii) too limiting since a method of an invariant-declaring class may need to invoke methods on the objects related by the invariant to re-establish the invariant.

Existing techniques [5–12] for verifying multi-object invariants differ in how they address the challenges mentioned above as well as in the range of verification problems they can handle. Given the challenges that real-world, object-oriented programs pose for invariant-based verification, it has been questioned whether the object invariant is the correct foundation for verifying object-oriented programs [15]. In this paper, we demonstrate that an object-oriented programming language with explicit support for *relationships* enjoys properties that facilitate the verification of real-world programs with invariants. Relationship-based programming languages [16–24] complement object-oriented programming languages with the programming language abstraction of a relationship to capture the interplay between objects. We introduce a visible states verification technique for Rumer, a simple relationship-based programming language developed to explore relationships [20]. The verification technique leverages the particular modularization properties of the Rumer language that we summarize as the “Matryoshka Principle”. The principle relies on a *stratification* of classes and relationships and stipulates restrictions on writes to fields, the expression of invariants, and method invocations. It translates into a stratification of invariants and guarantees the absence of transitive call-backs and, as a consequence, restores a visible states semantics for invariants. To facilitate modular reasoning about shared state, the verification technique leverages *member interposition* [20, 25] and *extent ownership*, two orthogonal language features supported in Rumer. We demonstrate our verification technique on the Composite pattern.

2 Running Example

The Composite pattern has been suggested recently as a verification challenge [12, 26] and served as the “Challenge Problem” for the 7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS). Figure 1(a) shows the class diagram of the Composite pattern. As indicated by the UML aggregation, the pattern “*composes objects into tree structures to represent part-whole hierarchies and lets clients treat leaf objects and composite objects uniformly*” [27]. Figure 1(b) shows a slightly modified version of the Composite pattern implementation presented in [26]. In accordance with the UML class diagram, the implementation distinguishes the classes Component and Composite and represents the UML aggregation by means of a parent reference and a children collection in Component and Composite, respectively. The implementation further establishes a field `total` (line 3 in Fig. 1(b)) which indicates the total number of children components that can be reached from a component.



```

1 class Component {
2   protected Composite parent;
3   protected int total = 0;
4 }
5
6 class Composite extends Component {
7   private Collection<Component> children;
8
9   public Composite() {
10    children = new Vector<Component>();
11  }
12
13   public void addComponent(Component c) {
14    children.add(c);
15    c.parent = this;
16    addToTotal(c.total + 1);
17  }
18
19   private void addToTotal(int incr) {
20    total = total + incr;
21    if (parent != null) {
22      parent.addToTotal(incr);
23    }
24  }
25 }
  
```

(b)

Fig. 1. Object-oriented implementation of the Composite pattern. (a) UML class diagram of the Composite pattern. (b) Implementation of the Composite pattern in a Java-like language as suggested in [26].

The Composite pattern gives raise to a number of interesting invariants. The SAVCBS 2008 challenge problem, in particular, stipulates the invariant that the value of a component object’s `total` field must be equal to the number of children components contained within the sub-tree rooted at the component object. This invariant is an instance of a multi-object invariant since it is violated

by any addition or removal of a component to or from a composite's sub-tree. Method `addComponent()` accounts for this violation and triggers a bottom-up traversal of the composite tree to update the total field of a composite and of all its parent composites. The actual update is achieved by the recursive method `addToTotal()`. Once this method terminates, the invariant of the composite on which `addComponent()` is invoked as well as the invariants of all its transitive parent composites will be re-established. However, for the duration of the recursive invocations of `addToTotal()`, those invariants are broken. Since these invocations (re-)enter inconsistent objects, method `addToTotal()` cannot assume that the invariant of its current receiver object holds in the initial state of the method.

The verification of the **Composite** pattern is challenging since it features a multi-object invariant and disallows a naive adoption of a visible states semantics. This restriction rules out, for instance, the Classical Technique [9] for verifying the **Composite** pattern since it can neither accommodate multi-object invariants nor re-entrant method invocations. A number of proposals to address the challenges of verifying the **Composite** pattern have been suggested. Verification techniques based on *ownership* [5, 7, 9] allow the specification and verification of the **Composite** pattern by leveraging the heap topology enforced by ownership types. However, an ownership-based specification of the **Composite** pattern prevents direct modifications of a composite's components. Other proposals typically employ a relaxed visible states semantics for invariants. Summers and Drossopoulou [12], for instance, introduce Considerate Reasoning, a verification technique that is based on a visible states semantics for invariants but allows distinguished invariants to be broken in the initial states of method executions provided that the methods re-establish the invariant in the final state. In addition, techniques have been presented that do not employ a visible states semantics for invariants. Bierhoff and Aldrich [28], for instance, leverage type-state-based permissions to verify a simpler invariant for binary **Composite** tree structures and Jacobs et al. [29] leverage separation logic to verify the SAVCBS invariant also for binary **Composite** tree structures.

In this paper, we show how first-class relationships allow for a precise specification of the **Composite** pattern. Our specification captures not only the SAVCBS invariant regarding a composite's `total` field but also gives a precise definition of a composite's tree properties. Using higher-level programming language abstractions and their stratification, we can encapsulate the multi-object invariant of the **Composite** pattern in a relationship and restore a visible state semantics for invariants.

3 First-Class Relationships

This section introduces the specification of the **Composite** pattern in Rumer and discusses the language features that are important for modular program verification. Rumer is a relationship-based programming language with Design-by-Contract-style [2] assertions. To gain practical experience with first-class relationships, we designed and implemented a prototype compiler that supports

the features shown in this paper and offers run-time checking of Design-by-Contract-style assertions. The Rumer compiler has been the basis for various student projects at ETH Zurich.

3.1 Language Principles

This section provides an overview of Rumer's basic language features. In the subsequent sections, we introduce each language feature in turn, based on the Rumer implementation of the Composite pattern shown in Fig. 2. The assertion language of Rumer is covered in Sect. 3.2.

Programmer-Definable Types. Figure 2 shows the implementation of the Composite pattern in Rumer. The program consists of three type declarations: the entity declaration `Component` and the relationship declarations `Parent` and `Composite`. An *entity* abstracts the state and behavior that a number of objects have in common. A *relationship* abstracts the state and behavior that a number of related objects have in common. Both language abstractions can be instantiated; we use the terms *entity instance* or *object* to denote instances of type entity and the term *relationship instance* to denote instances of type relationship. An entity resembles a class in a pure object-oriented language as it can define fields and methods. The existence of first-class relationships, however, fundamentally changes the position an entity takes in a relationship-based programming language. Using the abstraction of a relationship, a programmer can factor out the description of how objects relate into a relationship, rendering the need to establish references in an entity unnecessary. In Rumer, we build on this observation and prohibit the declaration of references in entities, requiring the declaration of how abstractions and their instances relate to happen exclusively in relationships. This language requirement results in a *stratification* of entities and relationships as only relationships know about their participating entities, but not vice versa. In Sect. 4 we discuss the benefits of the resulting stratification for program verification.

Simple Relationship Declaration. Relationships declare the types of instances they relate in their `participants` clause. For example, relationship `Parent` relates entity instances of type `Component` (line 3 in Fig. 2). Rumer allows programmers to associate an identifier with each type declaration in a `participants` clause to denote the *role* an instance of the type plays in the relationship. In case of relationship `Parent`, we use the role names `child` and `parent` to represent the hierarchical structure of the Composite pattern. Figure 3(a) provides a graphical illustration of a snapshot of a Rumer program heap comprising `Parent` relationship instances. The figure represents entity instances as dark gray circles and relationship instances as light gray ellipses. The heap snapshot consists of six `Component` entity instances and five `Parent` relationship instances. Each `Parent` relationship instance has a `Component` entity instance as `child` participant and a `Component` entity instance as `parent` participant. As indicated by the arrows in the figure, the role identifiers `child` and `parent` denote references from a `Parent` relationship instance to its participating `Component` objects.

```

1 entity Component {...}
2
3 relationship Parent participants (Component child, Component parent) {
4   int >parent total; // interposed instance field
5
6   extent void append(Component c, Component p) {
7     these.add(new Parent(c, p));
8     foreach (x isElementOf these.transitiveClosure() .select(c_p:
9       c_p.child == c).parent)
10    { x.total = x.total + 1; }
11  }
12
13 // A Composite relationship instance owns its tree Parent extent
14 relationship Composite participants (Component root, owned Extent<Parent>tree) {
15
16   extent void createComposite(Component c)
17   { these.add(new Composite(c, new owned Extent<Parent>())); }
18
19   void appendComponent(Component c, Component p)
20   { this.tree.append(c, p);}
21
22   void appendSubComposite(query Set<Parent> c, Component p) {
23     foreach (c_p isElementOf c .select(x: x.parent == p)) {
24       this.appendComponent(c_p.child, c_p.parent);
25       this.appendSubComposite(c .select(x: x.child isElementOf
26         c .transitiveClosure() .select(y: y.parent == c_p.child).child),
27         c_p.child);
28     }
29
30   void appendComposite(Composite c, Component p) {
31     this.appendComponent(c.root, p);
32     this.appendSubComposite(c.tree, c.root);
33   }

```

Fig. 2. Relationship-based implementation of the Composite pattern in Rumer

Member Interposition. Relationships (like entities) can declare *instance members* (i.e., instance fields and instance methods). For example, relationship `Parent` declares an instance field `total` (line 4 in Fig. 2). Unlike entity instance members, relationship instance members can either be associated with the relationship instance or with one of the relationship’s participant instances. The latter is achieved using the Rumer language mechanism *member interposition* [20, 25]. Member interposition allows a participant instance of a relationship instance to be “decorated” with additional fields and methods. Interposed relationship members are declared using the ‘>’ sign, which precedes the role identifier of the participant into which the member is interposed. In the example, the field `total` is interposed into the `Component` entity instance that acts as a parent in a `Parent` relationship instance. The field `total` is conceptually equivalent to the `total` field of the SAVCBS 2008 challenge problem (line 3 in Fig. 1(b)) as it allows a parent component to store the number of all children components it is directly or indirectly related to. In Fig. 3(a), the interposed relationship instance field `total` is displayed in the light gray arc that is attached to the `Component` entity instance to which the relationship instance refers by the `parent` reference. A non-interposed relationship instance field, on the other hand, would be displayed in the light gray ellipse representing the relationship instance. Like non-interposed relationship instance fields, interposed relationship

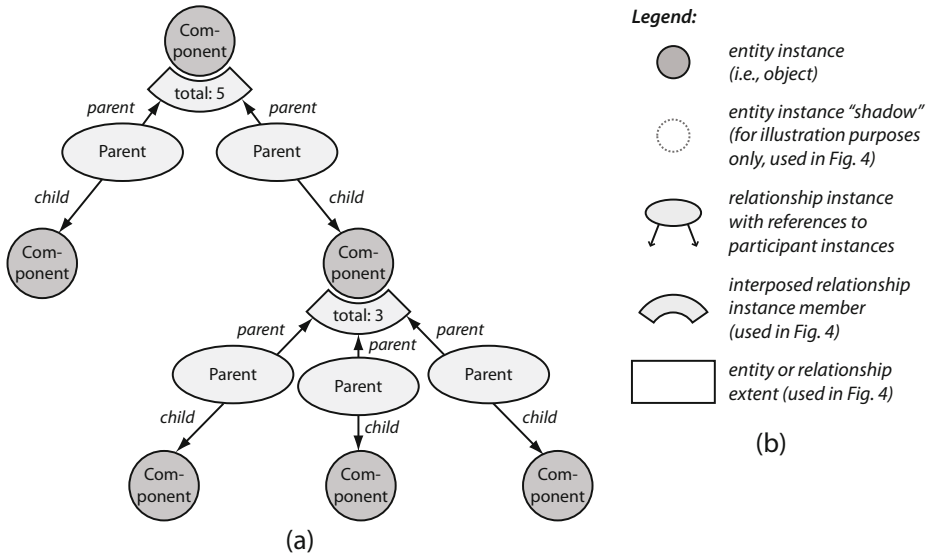


Fig. 3. (a) Schematic illustration of the Rumer program heap for the Parent relationship declared in Fig. 2 (b) Legend for Fig. 3(a) and Fig. 4

instance fields are fully encapsulated in the relationship that declares the field. As a result, interposed relationship instance fields are only accessible from instances of the declaring relationship but not from the participant instances into which the fields are interposed.

Nested Relationship Declaration and Extent Ownership. The declaration of the relationship `Composite` illustrates that relationships can have other relationships as participants. A `Composite` relationship instance relates a `Component` entity instance to an extent of the `Parent` relationship. Every Rumer entity or relationship declaration T has a corresponding *extent* type $\text{Extent}(T)$. An instance of an entity or relationship extent comprises a set of entity or relationship instances, respectively. Extents are explicitly instantiated as well as populated and depopulated with instances by the programmer. For example, a `Parent` extent is instantiated on line 117 in Fig. 2 (new owned `Extent<Parent>()`) and populated by invoking the built-in `add()` method on lines 7 and 17 in Fig. 2. The keyword `owned` in the participants clause establishes ownership of a `Composite` instance of the participating `Parent` extent. The ownership declaration requires the `Parent` extent to be instantiated and populated within the relationship and not to escape the relationship. As a result, the `Composite` instance becomes the unique owner of its associated `Parent` extent.

Figure 4 shows an extended version of the Rumer heap snapshot shown in Fig. 3 and displays all instances of the types declared in Fig. 2. The figure represents extents by rectangular boxes. The heap snapshot consists of one `Composite` extent, two `Parent` extents, and one `Component` extent. Each

extent comprises a number of instances. For example, the `Composite` extent comprises two `Composite` relationship instances, and the `Component` extent comprises four `Component` entity instances. `Component` instances can participate in the `Composite` relationship as well as in the `Parent` relationship. To keep the graphical layout well-arranged, Fig. 4 uses “shadow” instances. A shadow instance is depicted by a dotted circle and is a graphical copy of an actual instance to which it is connected by a dotted line. Each `Composite` relationship instance relates a `Component` entity instance to a `Parent` extent. The former is referred to by the role identifier `root` and the latter by the role identifier `tree`. In a relationship-based implementation a `Composite` is thus represented by a tuple that consists of a `root` component and a set of hierarchically structured components that represent the `tree` rooted at the `root` component. The `tree` of a composite may denote the empty set (if the composite only consists of one (leaf) component). The dotted line between a composite’s `root` component and the component at the top of the composite’s `tree` `Parent` extent indicate that the two components are indeed the same. The two dotted lines converging in the third `Component` instance in the `Component` extent illustrate that `Parent` instances of different extents can share `Component` instances. The sharing of `Component` instances among different `Parent` extents does not compromise the ownership declared for `Composite`. The ownership only encompasses a `Parent` extent but not the participating `Component` entity instances. This property distinguishes extent ownership from “traditional” ownership established by ownership types [30, 31] and Universe Types [5, 32] and distinguishes the `Rumer Composite` implementation from one based on the ownership technique [5, 9].

Instance and Extent Members. To populate and depopulate their extents, entities and relationships declare extent methods. An *extent method* has an implicit target, which denotes the extent on which the method is called. The keyword `these` refers to the implicit target extent. Extent methods are distinguished by the `extent` keyword, which precedes the method’s return type declaration. The `Composite` pattern implementation in Fig. 2 declares the relationship extent methods `append()` and `createComposite()`. The methods `appendComponent()`, `appendSubComposite()`, and `appendComposite()` are non-interposed relationship instance methods. In addition to extent methods, `Rumer` supports the declaration of extent fields (not used in Fig. 2). An extent field denotes the state of a whole extent, as opposed to an instance field, which denotes the state of an individual entity or relationship instance comprised in an extent.

Queries. `Rumer` provides *query expressions* (similar to LINQ heap queries [33]) to allow access to the instances contained in an extent. For example, method `append()` declares a query expression on line 8 (see Fig. 2) that makes use of the built-in query operators `transitiveClosure()` and `select()`. A query expression evaluates to a set that is constructed by invoking a query operator on a target extent or set. Whereas extents are explicitly instantiated and populated by programmers, sets can only be generated by querying extents or sets. In the example, the `transitiveClosure()` operator is invoked on the receiver extent of

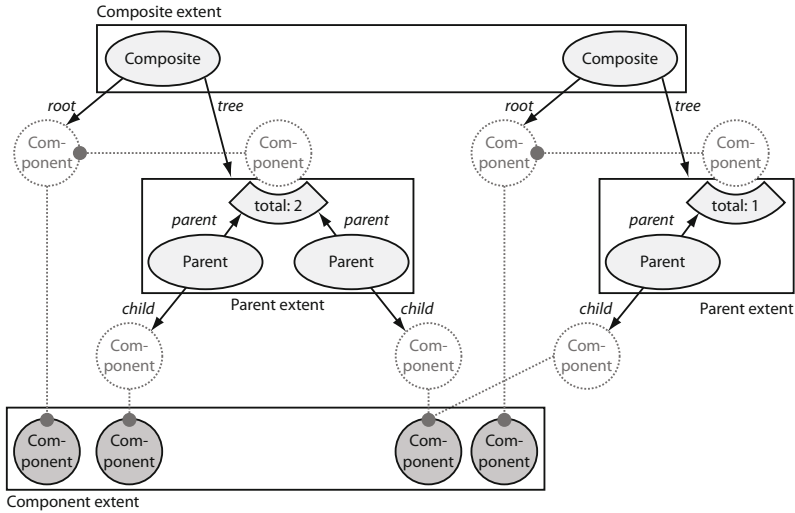


Fig. 4. Schematic illustration of the Rumer program heap for the implementation of the Composite pattern in Fig. 2 (see legend in Fig. 3(b))

the method `append()` and the `select()` operator is invoked on the set returned by the `transitiveClosure()` operator. The keyword `these` refers to the current receiver extent of an extent method invocation. The `transitiveClosure()` operator returns the transitive closure of its target set and the `select()` operator returns the subset of its target set that contains all the elements that satisfy the specified selection criterion. Like LINQ queries, the `select()` operator leverages lambda expressions to specify its selection criterion (i.e., `c.p: c.p.child == c`). As opposed to LINQ queries, Rumer queries are side-effect free. Side-effect freedom guarantees that the target sets of query operators are not altered in the course of the query evaluation and that query expressions become predicates over their target sets.

Implementation Details. Next we provide a brief overview of the individual method declarations in Fig. 2. These explanations are helpful to understand the details of the Composite pattern implementation in Rumer, however, are not a prerequisite to understanding the remainder of this paper. The impatient reader may continue with Sect. 3.2 and refer to this section as needed.

The extent method `append()` of relationship `Parent` appends the argument component `c` as child of the argument component `p`. To this end, the method instantiates a new `Parent` relationship instance with references to the components `c` and `p` and adds the new instance to the receiver extent of the method (line 7). The method `add()` is a language built-in method that adds the argument instance to the extent on which the method is invoked. The loop on line 8 increments the `total` field of all transitive parent components of the child component `c`. The loop header declaration uses built-in query operators, as discussed in the previous section.

The extent method `createComposite()` of relationship `Composite` instantiates a new `Composite` relationship instance and adds it to the current receiver extent of the method. The new instance has the component `c` as a root participant and an empty `Parent` extent as a tree participant.

The instance method `appendComponent()` of relationship `Composite` invokes the extent method `append()` with the argument components `c` and `p` on the current receiver relationship instance's `tree` extent. As a result, the component `c` is appended as a child of the component `p` in the composite's `tree` extent.

The instance method `appendSubComposite()` of relationship `Composite` appends the sub-composite denoted by the query expression `c` to the target composite as a child of component `p`. The method is implemented recursively to append the sub-composite in a depth-first traversal fashion. In each recursive invocation, one child component of the sub-composite is appended to its corresponding parent component in the target composite's tree extent. Recursion stops whenever the sub-composite `c` denotes the empty set. This is the case whenever a leaf component has been inserted in the preceding recursive invocation.

The instance method `appendComposite()` of relationship `Composite` appends the composite `c` to the target composite as a child of component `p`. The method first appends the `root` component of composite `c` to the target composite as a child of `p` and then invokes the method `appendSubComposite()` on the target composite to append the sub-composite rooted at `c`'s `root` component as a child of the previously inserted `root` component.

3.2 Assertion Language

The Rumer assertion language includes method *preconditions* and *postconditions*, *assert* statements, and *invariants*. Assertions can range over all abstractions available in the Rumer programming language. Invariants, in particular, can be declared both for type instances and type extents, giving rise to the following four invariant categories¹:

- **Entity instance invariant:** Property that must hold for each entity instance of the entity that declares the invariant.
- **Entity extent invariant:** Property that must hold for each extent instance of the entity that declares the invariant.
- **Relationship instance invariant:** Property that must hold for each relationship instance of the relationship that declares the invariant.
- **Relationship extent invariant:** Property that must hold for each extent instance of the relationship that declares the invariant.

Figure 5 lists the invariant declarations for the `Composite` pattern implementation in Fig. 2. The declaration consists of a relationship extent invariant for relationship `Parent` (line 3) and a relationship instance invariant for relationship `Composite` (line 12). Extent invariants are distinguished from instance invariants by the keyword `extent`. All invariant declarations in Fig. 5 adhere to the admissibility criteria defined in Sect. 4.2.

¹ These invariant categories refine the categories introduced in earlier work [20]

```

1 relationship Parent participants (Component child, Component parent) {
2   ... // See Fig. 2
3   extent invariant
4     these.isPartialFunction() &&
5     these.transitiveClosure().isIrreflexive() &&
6     forall(p isElementOf these.parent: p.total ==
7       these.transitiveClosure().select(c_p: c_p.parent == p).count());
8 }
9
10 relationship Composite participants (Component root, owned Extent<Parent>tree) {
11   ... // See Fig. 2
12   invariant
13     !(this.root isElementOf this.tree.child) &&
14     (!this.tree.isEmpty() => this.root isElementOf this.tree.parent) &&
15     this.tree.child == this.tree.transitiveClosure().select(c_p: c_p.parent ==
16       this.root).child;
17 }

```

Fig. 5. Invariant declarations for the Composite program in Fig. 2. See method preconditions and postconditions in Appendix A.

The extent invariant of relationship `Parent` leverages Rumer queries (see Sect. 3.1) and guarantees the following properties: (i) that every child component is related to at most one parent component (line 4), (ii) that the graph described by the `Parent` relationship is acyclic (line 5), and (iii) that the value of a parent component’s `total` field is equal to the number of children components to which the parent component is transitively related (line 6). Property (iii) satisfies the invariant of the SAVCBS 2008 challenge problem regarding a composite’s `total` field. Properties (i) and (ii) guarantee that the graph described by a `Parent` extent forms a forest of trees.

The instance invariant of relationship `Composite` restricts a composite’s `Parent` extent from a forest of trees to a tree by enforcing the following properties: (i) that a composite’s root component never appears as a child component in the graph described by the `Parent` extent (line 13), (ii) that a composite’s root component appears as a parent component in the graph described by the `Parent` extent unless the graph is empty (line 14), and (iii) that a composite’s root component is the parent component of all children components of the graph described by the `Parent` extent (line 15). These properties guarantee that a composite has a unique root and that a composite’s root component is the same as the one at the top of a composite’s tree. The heap snapshot shown in Fig. 4 represents a valid instantiation of the `Composite` pattern specification of Fig. 2 and Fig. 5. The shown `Composite` instances have unique roots and form trees. Note that the fact that different composites may share components (as indicated by the third shadow component in the `Component` extent) does not compromise the extent invariant of `Parent`. An extent invariant must hold for each extent instance but not for the union of all extent instances.

To guarantee the invariants declared in Fig. 5, the methods of the `Composite` pattern implementation (see Fig. 2) define preconditions and postconditions. The complete list of preconditions and postconditions for all methods is given in Appendix A. In the following, we highlight those preconditions and postconditions that are particularly interesting.

By appending the argument component c as child of the argument component p , method `append()` of relationship `Parent` may compromise the extent invariant of `Parent`. To prevent introducing cycles and relating a child component to several parent components, the method establishes the following precondition:

```
c != p && !(c isElementOf these.child union these.parent)
```

Furthermore, the method updates the `total` field of all transitive parent components of the child component c and thus ensures the following postcondition:

```
forAll(x isElementOf these.transitiveClosure().select(c_p: c_p.child == c).parent:
  x.total == old(x.total) + 1)
```

Method `appendComposite()` of relationship `Composite` appends the argument composite c to the target composite as a child of the argument component p . To prevent introducing cycles in the altered target composite, the method establishes the following precondition:

```
p != c.root && !(p isElementOf c.tree.child union c.tree.parent) &&
(this.tree.isEmpty() => p == this.root) &&
(!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent) &&
!(c.root isElementOf this.tree.child union this.tree.parent) &&
((this.tree.child union this.tree.parent) intersection
(c.tree.child union c.tree.parent)).isEmpty();
```

The above precondition would prevent us from appending the left composite instance of Fig. 4 to the right composite instance, or vice versa, since the last conjunct of the precondition could not be satisfied. The instance method `appendComposite()` invokes the instance method `appendSubComposite()` of relationship `Composite` for the actual insertion of the sub-tree rooted at c 's root component. To guarantee that the sub-tree c passed as argument indeed forms a tree with the root component p , method `appendSubComposite()` establishes the following precondition:

```
!(p isElementOf c.child) &&
(!c.isEmpty() => p isElementOf c.parent) &&
(c.child == c.transitiveClosure().select(c_p: c_p.parent == p).child) &&
```

The above precondition is equivalent to a composite's invariant. The postcondition of `appendSubComposite()`

```
this.tree == old(this.tree) union c;
```

precisely captures the “invariant” of the recursive implementation of the method that inserts the sub-tree c in a depth-first traversal fashion.

4 The Matryoshka Principle

The modularization discipline embodied in the Rumer programming language follows the “*Matryoshka Principle*”. We use the metaphor of the Russian nesting dolls to refer to the inherent *stratification* of programming language abstractions in Rumer. Based on this stratification, the principle defines admissibility criteria that stipulate restrictions on writes to locations, invariant declarations, and method invocations. We first introduce the stratification of the programming language abstractions. Then, we discuss the admissibility criteria.

4.1 Stratification

Figure 6 provides a schematic illustration of how programming language abstractions are stratified in Rumer. The figure shows an extended version of the Composite heap snapshot shown in Fig. 4 and depicts each programmer-defined type of the Composite program in addition to the extents and entity and relationship instances shown in Fig. 4. The stratification of the Rumer language abstractions is determined by the participants clauses of relationship declarations. Since only relationships can refer to their participants, but the participants not to their relationship, the participants clauses give rise to a *strict partial order* between relationships and participants. Figure 6 represents the resulting ordering of relationships and participants by placing relationships (relatively) above their participants. As indicated by the vertical arrows, the ordering makes a relationship become an *upper* layer of its participants, and conversely, the participants become a *lower* layer of their relationships.

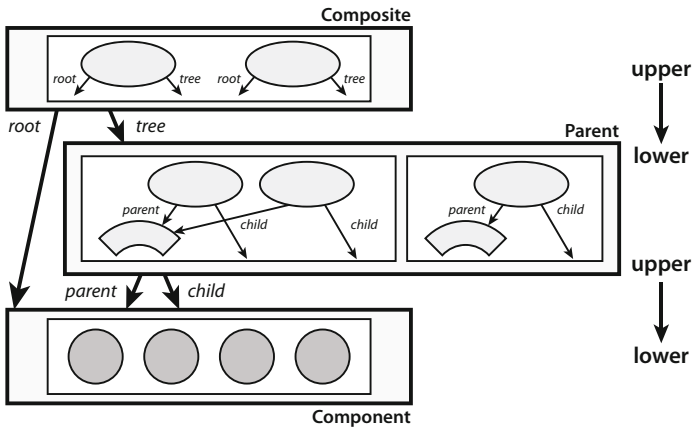


Fig. 6. Schematic illustration of the stratification of Rumer language abstractions (based on Fig. 4)

4.2 Admissibility Criteria

The admissibility criteria rely on the stratification of language abstractions shown in Fig. 6. The criteria stipulate restrictions that define (i) which methods are allowed to write to which locations, (ii) which invariants are allowed to depend on which locations, and (iii) on which instances a currently executing method can invoke another method.

In Rumer, a *location* can be an instance field, an extent field, or a whole extent. Table 1 lists all possible Rumer locations and indicates for each location by what program statement it can be written to. The admissibility criteria are formulated relative to the set of locations that can be reached by an instance. This set of locations is determined by the participants clauses of relationship

declarations which indicate which other instances a particular instance may reach. The general direction of “reachability” between instances conforms to the ordering of relationships and participants indicated by the arrows in Fig. 6. This ordering allows a relationship extent to reach itself and any of its participating extents and allows a relationship instance to reach itself and any of its participating instances. Orthogonal to the direction of reachability determined by participants clauses, an extent can reach any of the instances it comprises by formulating an appropriate query expression.

Table 1. Rumer locations and program statements that can write to them

Location		Write
Entity	instance field	Assignment to field.
	extent field	Assignment to field.
	extent	Invocation of <code>add()</code> or <code>remove()</code> on extent.
Relationship	interposed instance field	Assignment to field.
	non-interposed instance field	Assignment to field.
	extent field	Assignment to field.
	extent	Invocation of <code>add()</code> or <code>remove()</code> on extent.

Admissible Writes. To allow for modular verification, the Matryoshka Principle requires that a method writes only to a location that is reachable from the current receiver and that is declared by the same type as the method. This requirement guarantees that all Rumer locations are encapsulated in the types that declare the locations.

The assignment to the interposed instance field `total` of relationship `Parent` on line 10 in Fig. 2, for example, is admissible since `x` refers to a `parent` component of a `Parent` relationship instance residing in `these` and since the assignment occurs in a method declared by the same relationship (i.e., `Parent`) as the field `total`. The admissibility of the assignment also relies on the fact that interposed instance fields are treated as fields of the relationship instance even though they describe properties of relationship participants. In previous work [25], we have shown how member interposition facilitates modular reasoning over shared state at the example of the `Observer` pattern. The invocations of the built-in method `add()` on line 7 and line 17 in Fig. 2 represent admissible writes as well since they write to the current receiver `extent` and since they occur in `extent` methods of the types that declare the current receiver `extent`.

Admissible Invariants. To allow for modular verification, the Matryoshka Principle requires that an invariant depends only on those locations (*a*) that are encapsulated in the type that declares the invariant, or alternatively, on those locations (*b*) that are declared by a type that is owned by the type that declares the invariant.

Both invariant declarations in Fig. 5 are admissible. The relationship extent invariant of `Parent` depends on the current receiver `Parent` `extent` as well as on `Parent`’s interposed relationship instance field `total`. These locations are

encapsulated by the relationship `Parent`. The relationship instance invariant of `Composite` is admissible due to a `Composite`'s instance ownership of its `Parent` extent.

Admissible Method Invocations. To allow for a visible states semantics and inductive reasoning, a verification technique must either be guaranteed that call-backs do not occur or be in the position to statically identify those invocations that may result in a call-back. Prohibiting call-backs in general is not feasible since it would also prevent direct call-backs. A *direct call-back* occurs if an executing method invokes a method on the same receiver instance as the one of the executing method. Recursive method invocations are special instances of direct call-backs. Prohibiting recursive method invocations would be too limiting a restriction. Moreover, direct call-backs can be statically identified and guarded with the proof obligation to re-establish the invariant of the current receiver instance before the call. Transitive call-backs, on the other hand, cannot be statically determined but can only be over-approximated. A *transitive call-back* occurs if an executing method invokes a method on a different receiver instance as the one of the executing method and if the invoked method or any of the methods it transitively invokes calls back into the original receiver instance. In a pure object-oriented setting, call-backs are essential for re-establishing a multi-object invariant. In a relationship-based language, however, multi-object invariants can be expressed at the right level of abstraction, relieving the need of a call-back to re-establish a multi-object invariant.

To guarantee a visible states semantics for invariants, the Matryoshka Principle requires that a currently executing method can only invoke a method (a) on the same receiver instance as the currently executing method or (b) on a receiver instance that is of a lower type than the declaring type of the currently executing method. These requirements guarantee that method invocations are either recursive, propagate downwards, or are dispatched over an instance contained in an extent. As a result, transitive call-backs are prevented and a visible state semantics for invariants can be maintained.

All method invocations occurring in the `Composite` pattern implementation in Fig. 2 are admissible since they either constitute direct call-backs or invocations on a receiver instance that resides in a lower stratification layer or in the extent of the calling method.

5 Verification Technique

This section introduces the visible states verification technique for Rumer. Section 5.1 briefly introduces the unified framework for visible states verification techniques [34], which we use for presenting our technique. Section 5.2 details the proof obligations of our technique. We proved our verification technique to be sound in [35].

5.1 Background

To describe our verification technique, we use the unified framework² for visible states verification techniques introduced by Drossopoulou et al. [34]. The framework captures a verification technique in terms of seven *parameters*. These parameters are:

- \mathbb{X} invariants expected to hold in initial and final states of a method execution (i.e., visible states).
- \mathbb{V} invariants vulnerable to a method execution, i.e., which may be broken while the method executes.
- \mathbb{B} invariants that must be proven to hold before a method call.
- \mathbb{E} invariants that must be proven to hold in the final state (i.e., at the end) of a method execution.
- \mathbb{U} permitted receivers of field updates.
- \mathbb{D} invariants that may depend on a given heap location (and indirectly locations on which an invariant may depend).
- \mathbb{C} permitted receivers of method calls.

Figure 7 illustrates the meaning of the framework parameters for the method `appendComponent()` on line 19 in Fig. 2. As demonstrated by the figure, \mathbb{X} can be assumed to hold in the initial and final states of method `appendComponent()`. In between these visible states only $\mathbb{X} \setminus \mathbb{V}$ can be assumed to hold since some invariants may be temporarily broken by the execution of the method. For field updates and method calls, the receiver instances must be checked to be in \mathbb{U} and \mathbb{C} , respectively. For example, before the invocation of method `append()` on the receiver `this.tree` it must be checked that the instance referred to by `this.tree` is in \mathbb{C} . In the pre-state of a method call (i.e., `append()`), \mathbb{B} must be proven, and in the final state of the method execution (i.e., `appendComponent()`), \mathbb{E} must be proven. For assignments (not shown in Fig. 7), lastly, it must be checked that at most the invariants in \mathbb{D} are influenced.

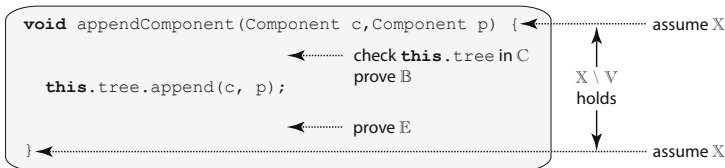


Fig. 7. Illustration of the verification technique framework parameters (based on [34])

² The framework has been introduced to capture visible states verification techniques for object invariants and to prove their soundness. Like Summers et al. [36], we use the framework for illustration purposes only.

5.2 Proof Obligations

Our verification technique for Rumer is a *visible states verification technique* [9]. It supports the complete assertion language discussed in Sect. 3.2. The technique is *modular* and allows entities and relationships to be verified independently from each other.

In this section, we describe our verification technique in terms of the seven parameters of the unified framework introduced previously. Since the parameters \mathbb{U} , \mathbb{D} , and \mathbb{C} are defined by the admissibility criteria introduced in Sect. 4.2, this section defines only the remaining parameters \mathbb{X} , \mathbb{V} , \mathbb{B} , and \mathbb{E} . We refer to the parameters \mathbb{X} , \mathbb{V} , \mathbb{B} , and \mathbb{E} as the *invariant parameters* since they specify sets of invariants, and to the parameters \mathbb{U} , \mathbb{D} , and \mathbb{C} as the *admissibility parameters* since they are captured by the admissibility criteria of the Matryoshka Principle. The verification technique determines the invariant parameters for all kinds of programmer-definable methods (i.e., entity instance method, entity extent method, interposed relationship instance method, non-interposed relationship instance method, and relationship extent method) as well as for constructors and the non-pure built-in methods `add()` and `remove()` (see Sect. 3.1). Built-in query operators (see Sect. 3.1) do not need to be considered by the verification technique since they are side-effect free.

Table 2 and Table 3 specify the invariant parameters for programmer-definable methods. Although there are variations between the invariant parameters for the different kinds of methods, there is a general schema that can be observed: The invariant parameters \mathbb{V} and \mathbb{X} are determined by the admissibility parameters \mathbb{U} (“Admissible Writes”) and \mathbb{D} (“Admissible Invariants”) and by the stratification of the programming language abstractions, respectively. The admissibility parameter \mathbb{U} guarantees that only the locations of the current receiver of a method can be written to. This admissibility parameter guarantees in turn that, while a method executes, at most the locations to which the method is allowed to write can change. The set of “vulnerable” locations indirectly determines the set of invariants \mathbb{V} that are vulnerable to a method: it is the set of invariants that may depend on those locations that may change during the execution of a method. The set of locations that an invariant may depend on is determined by the admissibility parameter \mathbb{D} . This set of locations is different for an invariant that is based on ownership compared to an invariant that is not based on ownership. In case of an ownership-based invariant, the set of locations that an invariant is allowed to depend on includes the locations of those lower-level types for which ownership is declared. In case of an invariant that is not based on ownership, the set of locations that an invariant is allowed to depend on includes only the locations of the invariant declaring type. The set of invariants \mathbb{X} that are expected to hold in the visible states of a method, on the other hand, is determined by the parameter \mathbb{V} and the stratification of the Rumer programming language. Irrespective of whether ownership is declared, a method can always expect all invariants of lower-level types to hold in its visible states. If the declaring type

Table 2. Invariant parameters \mathbb{X} , \mathbb{V} , \mathbb{B} , and \mathbb{E} for entity instance method, entity extent methods, interposed relationship instance methods, and relationship extent methods

Method	Parameter
Entity instance	\mathbb{X} : Entity instance invariant of current receiver. Invariants of all types for which the entity is not a transitive participant.
	\mathbb{V} : Entity instance invariant of current receiver. Entity extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	\mathbb{B} : Entity instance invariant of current receiver if direct call-back.
	\mathbb{E} : Entity instance invariant of current receiver. Preservation of entity extent invariant of current receiver's extent.
Entity extent	\mathbb{X} : All instance and extent invariants of the entity. Invariants of all types for which the entity is not a transitive participant.
	\mathbb{V} : Entity instance invariants of all instances in current receiver extent. Entity extent invariant of current receiver extent. Ownership-based invariants of upper-level types.
	\mathbb{B} : Entity instance invariants of all instances in current receiver extent and entity extent invariant of current receiver extent if direct call-back. Entity instance invariant of callee if entity instance method is called.
	\mathbb{E} : Entity instance invariants of all instances in current receiver extent. Entity extent invariant of current receiver extent.
Interposed relationship instance	\mathbb{X} : Invariants of all types for which the relationship is not a transitive participant.
	\mathbb{V} : Relationship instance invariants of all relationship instances that have current receiver as participant. Relationship extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	\mathbb{B} : -
	\mathbb{E} : Preservation of relationship instance invariants of all relationship instances that have current receiver as participant. Preservation of relationship extent invariant of current receiver's extent.
Relationship extent	\mathbb{X} : All instance and extent invariants of the relationship. Invariants of all types for which the relationship is not a transitive participant.
	\mathbb{V} : Relationship instance invariants of all instances in current receiver extent. Relationship extent invariant of current receiver extent. Ownership-based invariants of upper-level types.
	\mathbb{B} : Relationship instance invariants of all instances in current receiver extent and relationship extent invariant of current receiver extent if direct call-back. Relationship instance invariant of callee if relationship instance method is called.
	\mathbb{E} : Relationship instance invariants of all instances in current receiver extent. Relationship extent invariant of current receiver extent.

of a method is owned by an upper-level type, the execution of the method may compromise any ownership-based invariant of the owning type. However, if the declaring type of a method is not owned by an upper-level type, the execution of the method can only compromise invariants of its declaring type.

We illustrate the verification technique based on the Rumer implementation of the Composite pattern (see Fig. 2 and Fig. 5). Method `append()` of relationship `Parent` is a relationship extent method. According to Table 2, the parameter \mathbb{V} for a relationship extent method comprises the relationship instance invariants of all relationship instances in the current receiver extent as well as the relationship extent invariant of the current receiver extent. Since relationship `Parent` only declares an extent invariant, the parameter \mathbb{V} for method `append()` comprises the relationship extent invariant declared on line 3 in Fig. 5. This is also the invariant that the method must prove to hold in the final state of the method

Table 3. Invariant parameters \mathbb{X} , \mathbb{V} , \mathbb{B} , and \mathbb{E} for non-interposed relationship instance methods

Invariant	Parameter
Benign	\mathbb{X} : Relationship instance invariant of current receiver. Invariants of all types for which the relationship is not a transitive participant.
	\mathbb{V} : Relationship instance invariant of current receiver. Relationship extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	\mathbb{B} : Relationship instance invariant of current receiver if direct call-back.
	\mathbb{E} : Relationship instance invariant of current receiver. Preservation of relationship extent invariant of current receiver's extent.
Malign	\mathbb{X} : Relationship instance invariant of current receiver. Invariants of all types for which the relationship is not a transitive participant.
	\mathbb{V} : Relationship instance invariants of all relationship instances that have current receiver's participant(s) as participant(s). Relationship extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	\mathbb{B} : Relationship instance invariant of current receiver if direct call-back.
	\mathbb{E} : Relationship instance invariant of current receiver. Preservation of relationship instance invariants of all relationship instances that have current receiver's participant(s) as participant(s). Preservation of relationship extent invariant of current receiver's extent.

(parameter \mathbb{E}). Given the preconditions of the method (see Appendix [A](#)) and the fact that the method updates the `total` field appropriately, the method is able to prove the invariant. To sustain a visible states semantics, Table [2](#) further requires that any method invocations on the current receiver extent are guarded with the proof obligation to re-establish the relationship extent invariant before the invocation. Method `append()` invokes the built-in `add()` method on line [7](#). The built-in methods `add()` and `remove()` (not shown in Tables [2](#) and [3](#)) are treated differently than user-defined methods. Since no call-backs can result from built-in methods, callers do not have to re-establish their invariants before invoking a built-in method. As a result, method `append()` can invoke method `add()` without re-establishing its relationship extent invariant. As indicated by Table [2](#), method `append()` can expect the following invariants to hold in its visible states (parameter \mathbb{X}): all instance invariants and extent invariants of `Component` instances and `Component` extents, respectively, as well as all instance invariants and extent invariants of `Parent` instances and `Parent` extents, respectively. However, `append()` cannot expect the invariants of its upper-level type `Composite` to hold in its visible states.

Relationship `Composite` declares an extent method as well as non-interposed instance methods. The reasoning regarding the verification of the extent method is analogous to the one employed for method `append()` of relationship `Parent`. We highlight the important aspects of verifying non-interposed relationship instance methods. The invariant parameters for non-interposed relationship instance methods are defined in Table [3](#). The table distinguishes two kinds of non-interposed relationship instance methods: *malign* versus *benign*. The differentiation is due to the occurrence of interposed relationship instance fields in a relationship instance invariant declaration. A relationship instance invariant may relate interposed fields to non-interposed fields or relate interposed fields of different participants (category “malign” in Table [3](#)). Alternatively, a relationship

instance invariant may only depend on non-interposed fields (category “benign” in Table 3). The relationship instance invariant of relationship `Composite` is a benign invariant since there are no interposed fields declared by the relationship. The set of vulnerable invariants \mathbb{V} for a non-interposed relationship instance method of `Composite` consists only of `Composite`’ instance invariant. If `Composite` declared an extent invariant, that invariant would be vulnerable as well as the invariant may depend on instance fields. In its final state, a non-interposed relationship instance method must prove that the instance invariant of its current receiver holds and that it preserves the relationship extent invariant of its current receiver’s extent. The proof obligation of “invariant preservation” has been introduced in the context of object and class invariants in [11] and [36], respectively. It represents a weaker proof obligation as it does not require a method to *assert* that an invariant holds but to show that it does *not break* the invariant (provided that it held initially). Both for entity instance methods and relationship instance methods our verification technique requires the method to prove preservation of the extent invariant. This proof obligation accounts for the fact that an instance method may break an extent invariant, but may not be in the position to re-establish that invariant. If an instance method cannot show to preserve an extent invariant, its corresponding code must be captured in an extent method.

The invariant parameters for malign non-interposed relationship instance methods account for the fact that modifications of interposed relationship instance fields may compromise not only the invariant of the current receiver instance but also the invariants of all those relationship instances that have participants in common with the current receiver instance. This can be the case whenever an invariant relates an interposed relationship instance field with an interposed relationship instance field of another participant or relates an interposed relationship instance field with a non-interposed relationship instance field. The invariant parameters for such malign invariants are slightly different. Most importantly, their set of vulnerable invariants \mathbb{V} contains also the invariants of all the relationship instances that have participants in common with the current receiver instance. These invariants are also the ones that must be shown to be preserved in the final state of the non-interposed relationship instance method.

6 Discussion and Related Work

Our work builds on the visible states verification techniques developed for object invariants [5, 9, 11, 12] and introduces a verification technique for a relationship-based language that supports invariants for entities and relationships both at the instance and the extent level. The ownership technique [5, 9] is the visible states verification technique for object invariants that is most closely related to our work. Our verification technique resembles the ownership technique in two aspects: (i) it leverages heap stratification to prevent transitive call-backs and (ii) it facilitates modular reasoning about multi-object invariants. However, whereas the ownership technique is only composed of a single “ingredient” (i.e., Universe

Types [5, 32]), our verification technique unifies three orthogonal “ingredients” that can be combined in multiple ways. This customization of ingredients allows a programmer to “trade” imposed restrictions and supported guarantees.

The basic ingredient of our verification technique is the Matryoshka Principle. It enforces a stratification of language abstractions and guarantees the absence of transitive call-backs. As opposed to the ownership technique, the absence of transitive call-backs does not come at the price of a single ownership restriction! In a Rumer program conforming to the Matryoshka Principle, a participant of a relationship may be a participant of several relationships.

The Matryoshka Principle can be overlaid with member interposition to facilitate the modular verification of multi-object invariants. Similarly to the ownership technique, member interposition mitigates the adverse affect of aliases to shared state, but in a less restrictive way. As opposed to the ownership technique, member interposition does not prevent an instance from participating in other relationships but only prevents the interposed field from being accessible outside the relationship. Member interposition entails furthermore a slightly different semantics in terms of proof obligations than the ownership technique: Whereas ownership allows the declaration of invariant-compromising methods in owned objects as long as the owner does not invoke these methods, member interposition prevents the declaration of a method in a relationship that compromises the relationship invariant.

The Matryoshka Principle can also be overlaid with extent ownership to facilitate the modular verification of multi-object invariants. Extent ownership allows an owning type to impose an invariant on the owned extent. Similarly to the ownership technique, extent ownership enforces single ownership of the owned extent. However, as opposed to the ownership technique, extent ownership only encompasses an extent but not any (transitive) participant instances. As a result, those (transitive) participant instances can be modified by an arbitrary instance, including the extent owner. Extent ownership is thus more “lightweight” than “traditional” ownership since it relies on the Matryoshka Principle to prevent transitive call-backs.

The Matryoshka Principle can finally be overlaid both with member interposition and extent ownership. This setup was chosen for the `Composite` pattern. In the Rumer implementation of the `Composite` pattern, member interposition facilitates the verification of the `Parent` invariant, which includes the SAVCBS 2008 challenge problem invariant regarding a composite’s `total` field. The implementation leverages extent ownership, on the other hand, to verify the `Composite` invariant, which imposes a tree structure on a composite’s `parent` extent.

A number of techniques address the issue of object-oriented program verification in the presence of shared mutable state by leveraging heap partitioning. Parkinson and Bierman [15, 37, 38] introduce the ideas of separation logic [39] to Java. An alternative expression of separation is used in works on dynamic frames [40–42] where pure methods or ghost fields denote a set of locations. Assertions on the disjointness of such dynamic frames then facilitates heap-local reasoning. Parkinson’s and Bierman’s abstract predicates bear resemblance with

the invariants of our work. Similarly to a relationship invariant, an abstract predicate imposes consistency conditions on the object structures in the heap. However, abstract predicates do not entail an invariant semantics. This offers some flexibility to the programmer who does not need to adhere to a discipline, but sacrifices data type induction.

More distantly related is also the work on relationship-based programming languages [16–24]. For a summary of other approaches to relationship-based programming we refer to an earlier paper [20]. However, Rumer differs importantly from other relationship-based programming languages by its inherent stratification and by its support for Design-by-Contract-style assertions and invariants.

7 Conclusions

The verification of object-oriented programs remains a research issue. In this paper we discuss how relationships facilitate the verification of programs with multi-object invariants. Relationships impose a stratification of programming abstractions and consequently allow for local reasoning about multi-object invariants so that a modular verification of multi-object invariants is possible. The key concepts that allow for such local reasoning are (i) “member interposition” — properties (or fields) of a relationship participant that belong logically to the participant yet are encapsulated in the relationship instance, (ii) “extent ownership” — lightweight ownership of a relationship instance of its participant extent, and (iii) the Matryoshka Principle.

The “Matryoshka Principle” exploits the stratification layers of a program’s abstractions and defines admissibility criteria for writes to locations, invariant declarations, and method invocations. Programs that obey this principle can be verified using the simple approach outlined here. The programming language Rumer, which we use for illustration in this paper, adheres to this principle by design. However, the principle is not tied to this particular programming language. Programs in other programming languages can incorporate the principle as well (and could then be verified using this approach), but the responsibility to make the program obey the principle would fall either upon a programmer or some program development tool.

This paper reports on the benefits of including relationships in a programming language for program verification — as interest in tools and techniques to verify programs increases, we expect the idea of “relationships” as a way to express the interplay between objects to deserve serious consideration in mainstream programming languages.

Acknowledgments. We are grateful to: Sophia Drossopoulou for discussions on relationships and program verification; Gavin Bierman and Matthew Parkinson for discussions on the “relationship” between relationships and abstract predicates; Alexander J. Summers for exchange on the semantics of invariant preservation; James Noble for discussions on the “relationship” between extent ownership and “traditional” ownership; Reto Conconi, Nicholas D. Matsakis, and Albert Noll for their feedback; and the anonymous reviewers for their valuable comments.

References

1. Hoare, C.: Proof of correctness of data representations. *Acta Inf.* 1(4), 271–281 (1972)
2. Meyer, B.: *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs (1997)
3. Barnett, M., M. Leino, K.R., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) *CASSIS 2004*. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University (2006)
5. Müller, P.: *Modular Specification and Verification of Object-Oriented Programs*. LNCS, vol. 2262. Springer, Heidelberg (2002)
6. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. *Leino, K* 3(6), 27–56 (2004)
7. Leino, K.R. M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) *ECOOP 2004*. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
8. Barnett, M., Naumann, J.D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Kozen, D. (ed.) *MPC 2004*. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
9. Müller, P., Poetzsch-Heister, A., Leavens, G.T.: Modular invariants for layered object structures. *Sci. Comput. Program* 62(3), 253–286 (2006)
10. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: De Nicola, R. (ed.) *ESOP 2007*. LNCS, vol. 4421, pp. 80–94. Springer, Heidelberg (2007)
11. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Invariants for non-hierarchical object structures. *Electr. Notes Theor. Comput. Sci.* 195, 211–229 (2008)
12. Summers, A.J., Drossopoulou, S.: Considerate Reasoning and the Composite Design Pattern. In: Barthe, G., Hermenegildo, M. (eds.) *VMCAI 2010*. LNCS, vol. 5944, pp. 328–344. Springer, Heidelberg (2010)
13. Spitzzen, J.M., Wegbreit, B.: The verification and synthesis of data structures. *Acta Inf.* 4(2), 27–144 (1975)
14. Guttag, J.V.: Notes on type abstraction (version 2). *IEEE Trans. Software Eng.* 6(1), 13–23 (1980)
15. Parkinson, M.J.: Class invariants: The end of the road? In: *IWACO* (2007)
16. Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: *OOPSLA*, vol. 481, pp. 466–481. ACM, New York (1987)
17. Albano, A., Ghelli, G., Orsini, R.: A relationship mechanism for a strongly typed object-oriented database programming language. In: *VLDB*, pp. 565–575. Morgan Kaufmann, San Francisco (1991)
18. Bierman, G.M., Wren, A.: First-class relationships in an object-oriented language. In: Black, A.P. (ed.) *ECOOP 2005*. LNCS, vol. 3586, pp. 262–286. Springer, Heidelberg (2005)
19. Pearce, D.J., Noble, J.: Relationship aspects. In: *AOSD*, pp. 75–86. ACM, New York (2006)
20. Balzer, S., Gross, T.R., Eugster, P.T.: A relational model of object collaborations and its use in reasoning about relationships. In: Ernst, E. (ed.) *ECOOP 2007*. LNCS, vol. 4609, pp. 323–346. Springer, Heidelberg (2007)

21. Wren, A.: Relationships for Object-oriented Programming Languages. PhD thesis, University of Cambridge (November 2007)
22. Østerbye, K.: Design of a class library for association relationships. In: LCSD (2007)
23. Bodden, E., Shaikh, R., Hendren, L.: Relational aspects as tracematches. In: AOSD, pp. 84–95. ACM, New York (2008)
24. Nelson, S., Pearce, D.J., Noble, J.: First class relationships for OO languages. In: MPOOL (2008)
25. Balzer, S., Gross, T.R.: Modular reasoning about invariants over shared state with interposed data members. In: PLPV, pp. 49–56. ACM, New York (2010)
26. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.* 19(2), 159–189 (2007)
27. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
28. Bierhoff, K., Aldrich, J.: Permissions to specify the Composite design patterns. In: SAVCBS, pp. 89–94 (2008)
29. Jacobs, B., Smans, J., Piessens, F.: Verifying the Composite pattern using separation logic. In: SAVCBS, pp. 83–88 (2008)
30. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
31. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for exible alias protection. In: OOPSLA, pp. 48–64. ACM, New York (1998)
32. Dietl, W.: Universe Types Topology, Encapsulation, Genericity, and Tools. PhD thesis, ETH Zurich, 18522 (2009)
33. Bierman, G.M., Meijer, E., Torgersen, M.: Lost in translation: Formalizing proposed extensions to C#. In: OOPSLA, pp. 479–498. ACM, New York (2007)
34. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.: A Unified Framework for Verification Techniques for Object Invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
35. Balzer, S.: A relationship-based programming language and its value for program verification. Technical report, ETH Zurich (2011)
36. Summers, A.J., Drossopoulou, S., Müller, P.: Universe-type-based verification techniques for mutable static fields and methods. *JOT* 8(4), 85–125 (2009)
37. Parkinson, M.J.: Local Reasoning for Java. PhD thesis, University of Cambridge (2005)
38. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL, pp. 247–258. ACM, New York (2005)
39. O’Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
40. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
41. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
42. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)

A Pre- and Postconditions for Composite Specification

Below table indicates, for every method declared in Fig. 2, its preconditions and postconditions.

Precondition of Parent.append():
<code>c != null && p != null && c != p && !(c isElementOf these.child union these.parent) && (!these.isEmpty() => p isElementOf these.child union these.parent);</code>
Postcondition of Parent.append():
<code>these.count() == old(these.count()) + 1 && these.select(c_p: c_p.child == c).count() == 1 && forall(x isElementOf these.select(c_p: c_p.child == c): x.parent == p) && forall(x isElementOf these.transitiveClosure().select(c_p: c_p.child == c).parent: x.total == old(x.total) + 1);</code>
Precondition of Composite.createComposite():
<code>c != null;</code>
Postcondition of Composite.createComposite():
<code>these.count() == old(these.count()) + 1 && thereExists(x isElementOf these: x.root == c && x.tree.isEmpty());</code>
Precondition of Composite.appendComponent():
<code>c != null && p != null && c != p && (this.tree.isEmpty() => this.root == p) && (!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent && !(c isElementOf this.tree.child union this.tree.parent));</code>
Postcondition of Composite.appendComponent():
<code>this.tree.count() == old(this.tree.count()) + 1 && this.tree.select(c_p: c_p.child == c).count() == 1 && forall(x isElementOf this.tree.select(c_p: c_p.child == c): x.parent == p);</code>
Precondition of Composite.appendSubComposite():
<code>p != null && !(p isElementOf c.child) && !(c.isEmpty() => p isElementOf c.parent) && (c.child == c.transitiveClosure().select(c_p: c_p.parent == p).child) && (this.tree.isEmpty() => this.root == p) && (!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent) && (c.child intersection (this.tree.child union this.tree.parent)).isEmpty();</code>
Postcondition of Composite.appendSubComposite():
<code>this.tree.count() == old(this.tree.count()) + c.count() && this.tree == old(this.tree) union c;</code>
Precondition of Composite.appendComposite():
<code>c != null && p != null && p != c.root && !(p isElementOf c.tree.child union c.tree.parent) && (this.tree.isEmpty() => p == this.root) && (!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent) && !(c.root isElementOf this.tree.child union this.tree.parent) && !(c.tree.isEmpty() => ((this.tree.child union this.tree.parent) intersection (c.tree.child union c.tree.parent)).isEmpty());</code>
Postcondition of Composite.appendComposite():
<code>this.tree.count() == old(this.tree.count()) + 1 + c.tree.count() && this.tree == old(this.tree) union {(c.root, p)} union c.tree;</code>

Patterns of Memory Inefficiency

Adriana E. Chis¹, Nick Mitchell², Edith Schonberg², Gary Sevitsky²,
Patrick O’Sullivan³, Trevor Parsons¹, and John Murphy¹

¹ University College Dublin, Dublin, Ireland

{adriana.chis,trevor.parsons,j.murphy}@ucd.ie

² IBM T.J. Watson Research Center, Hawthorne, NY, US

{nickm,ediths,sevitsky}@us.ibm.com

³ IBM Software Group, Dublin, Ireland

patosullivan@ie.ibm.com

Abstract. Large applications often suffer from excessive memory consumption. The nature of these heaps, their scale and complex interconnections, makes it difficult to find the low hanging fruit. Techniques relying on dominance or allocation tracking fail to account for sharing, and overwhelm users with small details. More fundamentally, a programmer still needs to know whether high levels of consumption are *too* high.

We present a solution that discovers a small set of high-impact memory problems, by detecting patterns within a heap. Patterns are expressed over a novel *ContainerOrContained* relation, which overcomes challenges of reuse, delegation, sharing; it induces equivalence classes of objects, based on how they participate in a hierarchy of data structures. We present results on 34 applications, and case studies for nine of these. We demonstrate that eleven patterns cover most memory problems, and that users need inspect only a small number of pattern occurrences to reap large benefits.

Keywords: memory footprint, memory bloat, pattern detection, tools.

1 Introduction

In Java, applications can easily consume excessive amounts of memory [13]. We commonly see deployed server applications consume many gigabytes of Java heap to support only a few thousand users. Increasingly, as hardware budgets tighten, memory per core decreases, it becomes necessary to judge the appropriateness of this level of memory consumption. This is an unfortunate burden on most developers and testers, to whom memory consumption is a big black box.

We have spent the past two years working with system test teams that support a family of large Java applications. These teams perform extensive tests of applications, driving high amounts of load against them. While running these tests, they look at gross measures, such as maximum memory footprint. They may have a gut feeling that the number is high, but have little intuition about whether easy steps will have any measurable impact on memory consumption. Sizing numbers alone, whether memory consumption of the process, of the heap,

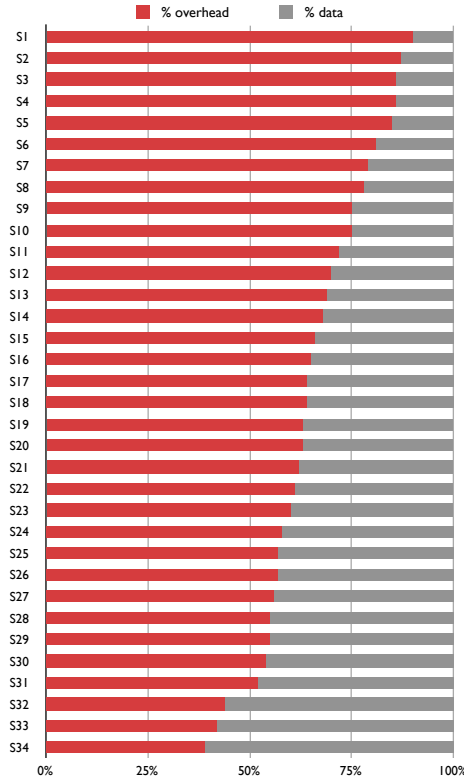


Fig. 1. The fraction of the heap that is overhead, including JVM object headers, pointers used to implement delegation, and null pointer slots, is often surprisingly high

and even of the size of individual data structures [3,18,10], are not sufficient. The test teams need a quick evaluation of whether deeper code inspections will be a worthwhile investment of time.

If size alone does not indicate appropriateness or ease of remediation, then perhaps measures of overhead can. Prior work infers an overhead measure, by distinguishing the *actual data* of a data structure from the implementation costs necessary for storing it in a Java heap [13]. Overheads come from Java Virtual Machine (JVM) object headers, null pointers, and various overheads associated with collections, such as the `$Entry` objects in a linked structure. Fig. 1 shows this breakdown, of actual data versus overhead, for 34 heap snapshots from 34 real, deployed applications. The figure is typically quite high, with most snapshots devoting 50% or more of their heap to implementation overheads.

Unfortunately, when presented with this figure, even on a per-structure basis, these testing teams were left with only a modified form of their original dilemma. Instead of wondering how large is too large, they now asked how much overhead is too much. To a development team, if a memory footprint problem is complicated to fix, or if the fix is difficult to maintain, it is of little value. Depending on the

nature of the overhead, a data structure may be easy to tune, or it may just be something the team has to live with. There are, after all, deadlines to be met.

An Approach Based on Memory Patterns. We have found that common design mistakes are made, across a wide range of Java applications. For example, it is common for developers to use the default constructors when allocating collections: `new ArrayList()`. If the code only stores a few items in these lists, we consider this to be an occurrence of a *sparse collection* pattern; only a few of the allocated pointers are used, thus the overhead comes from the empty slots. Hashmaps are often nested, and, if the inner maps are very small, this is an occurrence of the *nested small collection* pattern. These sound straightforward, and that was our goal: find problems that are easily understood, and easily fixed. Even if we miss large swaths of overhead, at least we are catching the easy stuff.

We discovered that this approach, when based on the right set of patterns, reliably explains a majority of overheads across a wide diversity of applications.

Detecting and Summarizing Pattern Occurrences. The challenges of this work came in two parts. First was the cataloging process. This involved a year of combing over data from many hundreds of real applications, to find the important patterns. The second challenge lay in the detection and reporting of pattern occurrences in complex heaps. Existing abstractions and analyses are insufficient to detect many of the common patterns. Knowing that items stored in a collection suffer from a high amount of delegation (a design that is actively encouraged [8]), with its attendant header and pointer overheads, requires knowing the *boundaries* of those items. Where would a scan of the graph of objects start, and where would it stop, in order to know that these items are highly delegated, and hence report the cost of this occurrence?

The choice of aggregation is crucial for detecting and reporting problems. Our approach is motivated by two properties prevalent in large-scale Java programs. First, multiple instances participate in larger cohesive units, due to the high degree of delegation common in the implementations of containers and user-defined entities. All of the objects in such a highly delegated design are grouped under a single occurrence of a larger pattern. We show how an aggregation by the *role* objects play in a data structure can ensure that we detect these larger patterns. Objects are either part of collection infrastructure, or part of the implementation details of contained items (entities or nested collections).

Furthermore, framework- and user-defined structures are frequently reused in multiple contexts. Frameworks themselves employ other frameworks, leading to deeply layered designs. Existing approaches aggregate by allocation context [7,16], or by only one hop of points-to context [2,18,3]. To generate meaningful reports, aggregation by deep context is important, in order to distinguish different uses (or misuses) of the same building blocks. The analysis cannot report millions of similar pattern occurrences, for example, one for each of the inner maps in a nest of hashmaps. In these cases, the context necessary to properly cluster can be in the dozens of levels of pointers.

The contributions of this paper are the following:

- Eleven commonly occurring patterns of memory misuse. Table 1 presents a histogram of the percentage of overhead explained by the eleven patterns, over 34 heap snapshots.
- The *ContainerOrContained Model*, a single abstraction that can be used both to detect occurrences of these patterns, and aggregate these occurrences into concise summaries based on the data structure context in which they occur. The abstraction defines the roles that data types play in the collection and non-collection implementation details of the application’s data models.
- An analysis framework for detecting and aggregating *pattern occurrences*, and encodings of the patterns as client analyses.
- A tool that implements this framework, and evaluations of its output on nine heaps. This tool is in initial use by system test teams within IBM.
- A characterization study of footprint problems in 34 heaps. The study shows, for example, that our set of patterns suffice to explain much of the overhead in heaps, and that a tool user typically need only inspect a small number of pattern occurrences to reap large benefits.

Fig. 2 summarizes our approach. We acquire a heap snapshot from a running Java application. From this snapshot, we compute the ContainerOrContained Model. We have encoded the patterns as clients of a common graph traversal algorithm. The traversal of a chosen data structure computes the count and overhead of each pattern occurrence, aggregated by its context within the structure.

2 The Memory Patterns

We have found that eleven patterns of memory inefficiency explain the majority of overhead in Java applications. The patterns can be divided into two main groups: problems with collections, and problems with the data model of contained items. The goal of this section is to introduce the patterns. In the next sections, we introduce a set of data structure abstractions and an algorithm for detecting and aggregating the *occurrences* of these patterns in a given data structure.

All of these patterns are common, and lead to high amounts of overhead. Table 2 names the eleven patterns. We use a short identifier for each, e.g. P1

Table 1. The analysis presented in this paper discovers easy to fix problems that quite often result in big gains. These numbers cover the heap snapshots in Fig. 1. The overhead is computed as described in Sect. 3.4.

overhead explained	# applications
0–30%	0
30–40%	4
40–60%	9
60–80%	7
80–100%	14

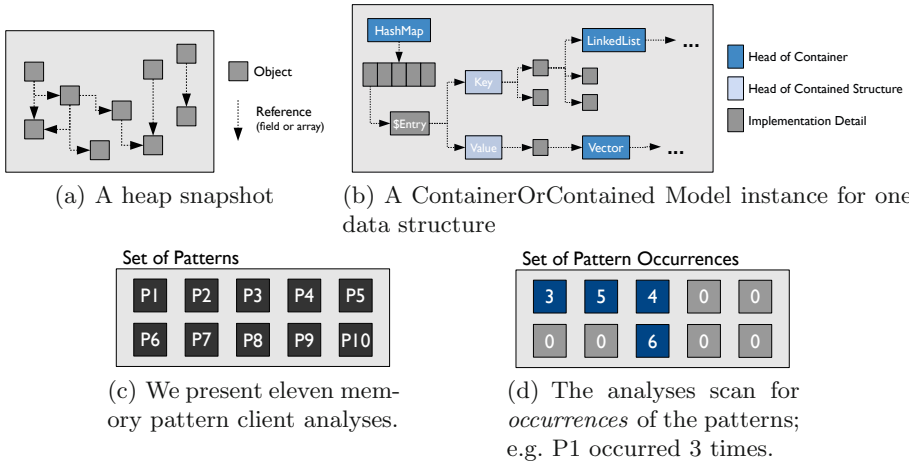


Fig. 2. From the raw concrete input, a heap snapshot from a Java application, we compute a set of abstract representations. We compute one abstract form, called the ContainerOrContained Model, per data structure in the heap snapshot. The client analyses scan each data structure for problematic memory patterns, making use of this abstract form.

stands for the pattern of empty collections. Table 3 shows that these patterns do indeed occur frequently across our sample heap snapshots, often multiple times per snapshot. Sect. 5 gives detailed findings of our detection algorithm.

2.1 Patterns P1–P3: Empty, Fixed, Small Collections

Each of these patterns has the general nature of a large number of collections with only a few entries. This situation leads to a high amount of overhead due to a lack of amortization of the fixed costs of a collection. The fixed costs of a `HashSet` in the Java standard library, which includes multiple Java objects and many field slots, is around 100 bytes (on a 32-bit JVM). This sounds like an inconsequential number, but if that set contains only a few entries, then the relative contribution of that fixed overhead to the total heap consumption is high. The fixed cost of a `ConcurrentHashMap` in Java is 1600 bytes!

Two important special cases have very different remedies from the general case of small collections. The first is the fixed-size small collections pattern, where all the instances of such collections contain always the same constant number of entries. These may benefit from using array structures, rather than a general purpose collection. The second is the empty collections pattern; occurrences of these could be lazily allocated.

2.2 Pattern P4: Sparsely Populated Collections

Collections that have an array-based implementation risk being sparsely populated. Fig. 3 shows an `ArrayList` that was instantiated with its default size,

Table 2. The eleven memory patterns

memory pattern	identifier
Empty collections	P1
Fixed-size collections	P2
Small collections	P3
Sparsely populated collections	P4
Small primitive arrays	P5
Boxed scalar collections	P6
Wrapped collections	P7
Highly delegated structures	P8
Nested Collections	P9
Sparse references	P10
Primitive array wrappers	P11

typically 10 or 12 entries, but that currently contains only two Strings. Unlike the first three patterns, this pattern affects both a large number of small (sparse) collections, and a single large (sparse) collection. The causes of a poorly populated collections are either: 1) the initial capacity of the collection is too high, or 2) the collection is not trimmed-to-fit following the removal of many items, or 3) the growth policy is too aggressive.

2.3 Pattern P5: Small Primitive Arrays

It is common for data structures to have many small primitive arrays dangling at the leaves of the structure. Most commonly, these primitive arrays contain string data. Rather than storing all the characters once, in a single large array, the application stores each separate string in a separate `String` object, each of which has its own small primitive character array. The result is often that the overhead due to the header of the primitive character array (12 bytes, plus 4 bytes to store the array size) often dwarfs the overall cost of the data structure. If this data is intended to be long-lived, then it is relatively easy to fix this problem. Java `Strings` already support this substring optimization.

Table 3. Across the 35 snapshots in Fig. 1, the memory patterns occur frequently. The patterns are also not spurious problems that show up in only one or two applications; many occur commonly, across applications.

	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11
# pattern occurrences	37	16	45	11	7	19	2	111	5	5	46
# applications	18	12	20	8	6	13	2	29	3	4	19

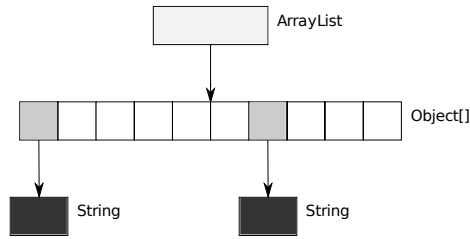


Fig. 3. An example of the sparse collection pattern, in this case with eight null slots

2.4 Pattern P6: Boxed Scalar Collections

The Java standard collections, unlike C++ for example, do not support collections with primitive keys, values, or entries. As a result, primitive data must be boxed up into wrapper objects that cost more than the data being wrapped. This generally results in a huge overhead for storing such data.

2.5 Pattern P7: Wrapped Collections

The Java standard library requires the use of wrappers to modify the behavior of a collections. This includes, for example, making a collection synchronized or unmodifiable. `HashSet` is implemented in this way, too: as a wrapper around `HashMap`. This is another case of a cost that would be amortized, if the collections had many entries, but one with a distinct remedy.

2.6 Pattern P8: Highly Delegated Structures

Java data models often require high degrees of delegation. For example, an employee has attributes, such as a name and email address. In Java, due to its single-inheritance nature, one is often forced to delegate the attributes to side objects; for example, the developer may wish to have these two attributes extend a common `ContactInformation` base class. The frequent result is a highly delegated web of objects, and the repeated payment of the object “tax”: the header, alignment, and pointer costs.

2.7 Pattern P9: Nested Collections

This pattern covers the common case of nested collections. One can use a `HashMap` of `HashSets` to model a map with multiple values per key. Similarly, a `HashMap` of `ArrayLists` can be used to represent a map which requires multiple objects to implement a key. Fig. 4 portrays a `HashMap` of `HashSet` where `String` key maps to a set of values, implemented using a `HashSet`. For this current paper, we only cover these two important cases of nested collections: `HashMaps` with either `HashSet` or `ArrayList` keys or values.

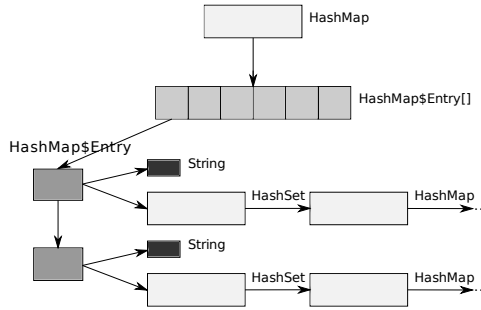


Fig. 4. An example of the HashMap of HashSet collection pattern

2.8 Pattern P10: Sparse References

In addition to high degrees of delegation, non-collection data often suffers from many null pointers. This pattern is an important special case of pattern P8: highly delegated structures. A highly delegated design can suffer from *overgenerality*. The data model supports a degree of flexibility, in its ability to contain extra data in side objects, that is not used in practice.

2.9 Pattern P11: Primitive Array Wrappers

The last non-collection overhead pattern comes from wrappers around primitive arrays. These include the `String` and `ByteArrayOutputStream` objects whose main goal is to serve as containers for primitive data. This is a cost related to P5: small primitive arrays, but one that is outside of developer control; hence we treat it separately. The Java language does not allow primitive arrays to be stored inline with scalar primitive data.¹ We include this pattern for completeness, even though in practice developers would have trouble implementing an easy fix to its occurrences. We wanted to include it, so that the characterization of Sect. 6 can motivate language and compiler developers to fix this problem.

3 The ContainerOrContained Abstraction

We introduce a single abstraction that is useful for both detecting occurrences of memory patterns and aggregating those occurrences in a way that concisely summarizes the problems in a data structure. We begin by describing the raw input to the system of this paper, and briefly present the important limitations of the dominator relation for heap analysis.

¹ Java supports neither structural composition nor value types, those features of C and C# that permit a developer to express that one object is wholly contained within another. At least it can be done manually, in the case of scalar data. This is simply not possible for array data.

3.1 Heap Snapshots and the Object Reference Graph

The system described in this paper operates on snapshots of the Java heap. This data is readily available from most commercial JVMs. Typically, one sends a signal to a Java process, at which point the JVM produces a file that contains the contents of the Java heap. The analysis can be done disconnected from any test runs, and the generated snapshots can be shared with development teams at a later date.

A heap snapshot can be considered as a graph of objects and arrays, interconnected in the way that they reference each other. Fig. 2(a) shows a small-scale picture of how we consider the heap to be a graph. This graph is commonly referred to as an *object reference graph*.

3.2 Limitations of the Dominator Relation

Several existing tools [3,18] base their visualizations and analyses on the *dominator forest* [9], rather than the full graph. This was also our first choice; it has some enticing qualities. When applied to an object reference graph, the dominator relation indicates unique ownership: the only way to reach the dominated object is via reference chasing from the dominator.

Unfortunately, the dominator forest is a poor abstraction for memory footprint analysis, due to the issue of shared ownership. Fig. 5 illustrates a case where two data structures share ownership of a sub-structure. An analysis that requires counting the number of items in a collection, such as the linked-list style structure in Data Structure 1, must count all items, whether or not they are simultaneously part of other structures. A traversal of the dominator tree of Data Structure 1 will only count two items — the edge leading to the third is not a part of the dominator forest. In addition to failing to account for paths from multiple roots, the dominator relation also improperly accounts for diamond structures.

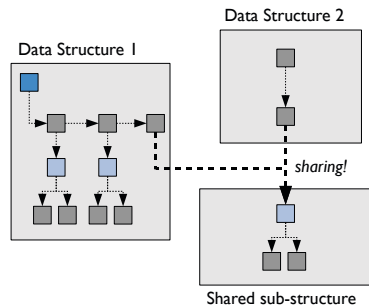


Fig. 5. The dominator forest is unhelpful both for detecting and for sizing memory problems. No traversal of a dominator tree (e.g. Data Structure 1 or 2) will encounter shared sub-structures. A collection, even if it dominates none of its constituents, should still be considered non-empty.

The dominator relation introduces a fake edge, from the root of the diamond to the tail. For these reasons, we base our analysis upon traversals of the full graph.²

3.3 Roles, and the ContainerOrContained Model

The bulk of objects in a data structure takes on one of six roles. These roles are summarized in Table 4. Consider a common “chained” implementation of a hashmap, one that uses linked list structures to handle collisions. Instances of this hashmap, in Java, are stored in more than one object in the runtime heap. One of these will be the entry point to the collection, e.g. of type `HashMap`, and the rest of the objects will implement the details of the linking structure. These two roles, the *Head of Container*, and *Implementation Details*, are common to most implementations of structures that are intended to contain an indefinite number of items.

Underneath the chains of this hashmap will be the contained data structures. These constituent structures have a similar dichotomy of roles: there are the *Head of Contained* structures, and, for each, the implementation details of that contained structure. Consider the example from earlier (Sect. 2.6): an `Employee` data structure that has been implemented to delegate some of its functionality to other data types, such as `PhoneNumber` and `EmailAddress`. That these latter two pieces of information have been encoded as data types and hence (in Java) manifested as objects at runtime, is an implementation detail of the `Employee` data structure. Another role comes at the interface between the container’s implementation details and the head of the contained items. For example, in a chained hashmap, the “entry” objects (`HashMap$Entry` in the Java standard collections library) will serve the role as this *Container-Contained Transition* objects. This role is crucial to correctly detect some of the patterns (shown in Sect. 4). The final important role, *Points to Primitive Array*, corresponds to those objects that serve as wrappers around primitive arrays.

We introduce the `ContainerOrContained` abstraction, that assigns each object in a data structure to at least one of these six roles. Objects not stored in a collection are unlikely to be the source of memory problems, and hence do not receive a role in this model. Given an object that is at the root of a data structure, we show how to compute that structure’s `ContainerOrContained` model.³ First, data types that form linking structures, such as the “entry” objects in a chained hashmap, are identified by looking for cycles in a points-to graph over types (a simple technique first described in [10]). Any instances of these types are assigned the *Transitory* role, and objects they reference are assigned the *Head of Contained* role. Any arrays of references that don’t point to a *Transitory* object are themselves *Transitory*; any objects these

² Many tools, including the authors’ previous work, made a switch over to using the dominator relation. The demos for one [3] even claim it as their “secret sauce”.

³ In the case that there is a connected component at the root of the data structure, choose any object from that cycle.

arrays point to are Heads of Contained structures. Finally, objects that point to arrays of references or recursive types are Heads of Containers. Note that it is possible for an object to serve multiple roles, simultaneously. A `HashMap` inside a `HashMap` is both Head of Container and Head of Contained. A `String` key is both Head of Contained, and it Points to a Primitive Array. The remaining objects are either the Implementation Details of a collection, or of the contained items.

Table 4. In the ContainerOrContained abstraction, objects serve these roles

Role	Examples
Head Of Container	<code>HashMap</code> , <code>Vector</code>
Head Of Contained	keys and values of maps
Container-Contained Transition	<code>HashMap\$Entry</code>
Points to Primitive Array	<code>String</code>
Collection Impl. Details	<code>HashMap\$Entry[]</code>
Contained Impl. Details	everything else

3.4 How Roles Imply Per-Object and Total Overhead

The ContainerOrContained model defines a role for each object in a data structure. Given this mapping, from object to role, we show that one can compute the *total overhead* in that data structure; previous work [13] introduced this concept, and here we show a novel, much simpler way, to approximate total overhead using only a ContainerOrContained model that doesn't rely on dominance. The goal of this paper is to explain as much of that total overhead as possible, with a small number of pattern occurrences.

Definition 1 (Per-object Overhead, Total Overhead). *Let G be an object reference graph and $D \subseteq G$ be a data structure of G . The total overhead of D is the sum of the per-object overhead of each object in D . The per-object overhead of an object depends on its role:*

- **Entirely overhead:** *if its role is Head of Container, Transitional, or Collection Implementation Detail, then the portion is 100%.*
- **Headers and pointers:** *if its role is Head of Contained or Contained Implementation Detail, then the portion includes only the JVM headers, alignment, and pointer costs.*
- **Headers, pointers, and primitive fields:** *if its role is Points to Primitive Array, we also include primitive fields, under the assumption that many primitive array wrappers need to store bookkeeping fields, such as offsets, lengths, capacities, and cached hashcodes, that are not actual data.*

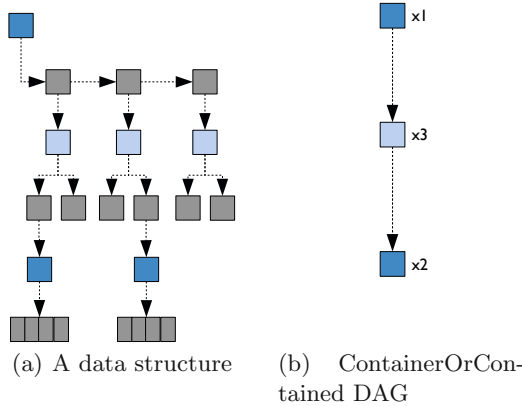


Fig. 6. A data structure, and its corresponding ContainerOrContained DAG

3.5 Regions, and the ContainerOrContained DAG

For every data structure, and the ContainerOrContained model over its constituent types, there is a corresponding directed acyclic graph (DAG) that summarizes the structure’s contents [13]. Informally, the ContainerOrContained DAG is one that collapses nodes in an object reference graph according to the role they play and the context in which they are situated. Fig. 6 shows an example data structure and the corresponding ContainerOrContained DAG. Observe how this structure has a two-level nesting of collections: the outer collection has a linking structure, and the inner map has an array structure. Sandwiched between the two collections is a contained item that delegates to two sub-objects; one of the sub-objects contains the inner array-based collection. The ContainerOrContained DAG collapses this structure down to a, in this case, tree with three nodes. In general, this summarized form will be a DAG, in the case of diamond structures.

We define the ContainerOrContained DAG according to an equivalence relation of object reference graph nodes. We present a novel definition and construction algorithm that shows how this DAG is directly inferrable from the ContainerOrContained model, without reliance on dominance. First, the nodes of a DAG are *regions*, which represent one of the two primary roles:

Definition 2 (Head of Region). *Let G be an object reference graph and C be a ContainerOrContained model of the types of G . We say that a node $n \in G$ is a Head of Region if the role of n under C is either Head of Container or the Head of Contained.*

From this, the equivalence relation is defined as follows:

Definition 3 (ContainerOrContained Equivalence). *Let G be an object reference graph, $D \subseteq G$ be a data structure in G with back edges pruned, and C*

be the *ContainerOrContained* model for the types of G . Two nodes $n_1, n_2 \in D$ are equivalent under C if either 1) n_1 is a head of region n_3 and n_2 is not and there is no intervening head of region n_3 between n_1 and n_2 (or vice versa for n_1 and n_2); or 2) neither n_1 nor n_2 is a head of region, but they lie under an n_3 that is a head of region, with no intervening head of region n_4 that is on either the path from n_1 to n_3 or from n_2 to n_3 ; or 3) n_1 and n_2 have the same role under C and the parents of n_1 are equivalent to the parents of n_2 .

4 Detecting Pattern Occurrences, and Aggregating Them by Context

We describe an algorithm that, parameterized by the details of a pattern, scans for occurrences of that pattern in a given data structure.⁴ We initially coded each pattern as its own set of code, but eventually came to realize that each pattern differed in only three ways:

- The **start** and **stop** criteria. The boundaries of an occurrence, the details of which vary from pattern to pattern, but can always be expressed in terms of roles. For example, whether a `HashMap` instance is an occurrence of the empty collection pattern depends on the objects seen in a traversal of the subgraph bounded on top (as one traverses from the roots of the data structure being scanned) by a Head of Container, and bounded on the bottom by the Heads of Contained items.
- The **accounting** metrics. Each pattern differs in what it needs to count, as the scan proceeds. The empty collection pattern counts the number of Heads of Contained. The sparse references pattern counts a pair of numbers: the number of valid references, and the number of null slots.
- The **match** criterion. The empty collections pattern matches that `HashMap` if the number of Heads of Contained objects encountered is zero.

Observe that the empty collections pattern cannot count the number of Transitional objects, (`HashMap$Entry` in this case), for two important reasons: 1) because some collections use these Transitional objects as sentinels; and 2) sharing may result in two Transitional objects referencing a single Head of Contained object.

Each match of a pattern would, without any aggregation, result in one pattern occurrence. This would result in needlessly complicated voluminous reports. Instead, as the algorithm traverses the data structure, it detects which *ContainerOrContained* region it is current in. Any occurrences of a pattern while the traversal is in a particular region are aggregated into that region. The output is a set of encountered regions, each with a set of occurrences. Each occurrence will be sized according to the accounting metrics of the pattern. Fig. 7 gives Java-like pseudocode for the algorithm.

⁴ A set of patterns can be scanned for simultaneously. The description in this section can be generalized straightforwardly to handle this.

For example, a scan for occurrences of the empty collections pattern would count the total overhead of the collection's Implementation Details, matches if the number of Heads of Contained is zero, and upon a match accumulate that overhead into the ContainerOrContained region in which the empty collection is situated.

4.1 Client Analyses

Each pattern is detected by a client analysis of the general algorithm of Fig. 7. In this paper, we chose three patterns that were illustrative of the interesting variations from client to client. Sect. 5 then describes an implementation, in which all eleven clients have been implemented.

P3: Small Collections. This client is activated at a Head of Container and deactivated at a Head of Contained. The client counts the per-object overhead of the collection's Implementation Details and the number of Head of Contained encountered. The accounting operation plays an important role in the pattern detection. In the case of a map, unless done careful, the client would double count the key and the value (recall that the Transitionary element points to a key and a value). We do so by deactivating the accounting when a Head Of Contained is reached. The scanning is reactivated at the traversal in postorder of the Transitionary object. A collection instance, e.g. one instance of a `HashMap`, matches the pattern if it contains, in the current implementation, at most nine entries.

P4: Sparsely Populated Collections. The scope of a sparsely populated collection is from a Head of Container element up to the next Head of Container or Head of Contained element. The pattern counts the number of null slots and its corresponding overhead and the number of non-null slots. The pattern matches the collection when the number of null slots is greater than the number of non-null slots.

P5: Small Primitive Arrays. The boundary of a small primitive array pattern occurrence is a Points to Primitive Array elements as start condition and the traversal stops when the primitive array is reached. The client counts the per-object overhead of the primitive array and the size of actual primitive data. A match happens when the amount of data is small, compared to the overhead.

Table 5 presents the time to detect⁵ all the eleven pattern categories in the heap snapshots from Fig. 1. While computation time is sometimes less than 2 minutes for heaps with tens of million of objects (e.g. Applications S8 and S9, with 57 and 34 million of objects respectively), there are also extreme cases where the computation time is high. Currently we are working on a few optimization possibilities to address the slow analysis time.

⁵ Using Sun's Linux JVM 1.6.0_13 64-Bit with `-server` flag, on a 2.66GHz Intel(R) Core(TM)2 Quad CPU.

```

interface Pattern {
    boolean start(Role role);
    boolean stop(Role role);
    boolean match(Accounting accounting);
    Accounting makeAccounting();

    interface Accounting {
        void accountFor(GraphNode node,
            Role roleOfNode);
        int overhead();
    }
}

interface PatternOccurrence {
    void incrOverhead(int bytes)
    void incrCount();
}

interface Region extends Map<Pattern,
    PatternOccurrence> {}

interface ContainerOrContainedModel {
    Role role(GraphNode node);
    Region equivClass(GraphNode node);

    enum Role { ... }; // see Table 4
}

Set<PatternOccurrences>
computePatternOccurrences(Pattern
    pattern, Graph dataStructure,
    ContainerOrContainedModel CoC) {
    Set<PatternOccurrences> occurrences; //
        the output

    dataStructure.dfsTraverse(new GraphData.
        Visitor() {
            Stack<Accounting> accountingStack;
            boolean active;

            void preorder(GraphNode node) {
                Role role = CoC.role(node)
                if (pattern.stop(role) {
                    if (!patternStack.isEmpty()) {
                        accountingStack.top().
                            accountFor(node, role
                                );
                        active = false;
                    }
                } else if (pattern.start(role)) {
                    active = true;
                    patternStack.push(pattern.
                        makeAccounting());
                }
                if (active) {
                    accountingStack.top().
                        accountFor(node, role)
                }
            }
            void postorder(GraphNode node) {
                if (pattern.start(node)) {
                    region = regionStack.pop();
                    if (pattern.match(
                        accountingStack.pop()) {
                        Region region = CoC.
                            equivClass(node);
                        PatternOccurrence occ =
                            region.get(pattern);
                        occ.incrCount();
                        occ.incrOverhead(
                            accountingState.
                                overhead());
                        occurrences.add(occ);
                    }
                } else if (pattern.stop(node)) {
                    active = true;
                }
            }
        });
    return occurrences;
}

```

Fig. 7. The algorithm that, given a pattern and a data structure, produces a set of aggregated pattern occurrences

5 Experiences with Our Tool

We have implemented the algorithm described in Sect. 4 and the eleven patterns in a tool that analyzes a heap snapshot for problematic uses of memory. It is in initial use by system test teams within IBM. The tool presents a list of the pattern occurrences, ranked by the amount of overhead they explain, and grouped by ContainerOrContained region. We have an initial visualization tool, under development, that will present these figures on a picture of the ContainerOrContained DAG. In this section, we walk through uses of the tool on nine of the heap snapshots from Fig. 1, to give a qualitative feeling of the value of the analysis, and to demonstrate the kinds of problems that routinely show up in

Table 5. The number of objects and computation time to detect all the patterns in the heap snapshots from Fig. 1.

	# objects [million]	time [minutes]	# objects [million]	time [minutes]	# objects [million]	time [minutes]		
S32	102	12.56	S27	14	1.59	S11	3.76	0.13
S4	58	114.37	S13	13	100.63	S25	3.09	7.98
S8	57	1.96	S18	12	3.71	S34	2.03	0.64
S3	50	13.59	S6	11	7.25	S24	1.82	8.78
S16	49	5.68	S1	11	1.52	S5	1.73	0.14
S17	37	19.8	S21	10	22.12	S29	1.41	0.73
S2	37	8.2	S23	8.29	6.06	S12	1.41	0.51
S26	36	9.75	S7	7.77	45.86	S30	1.37	3.14
S14	34	275.62	S15	5.83	77.21	S22	1.14	1.49
S9	34	1.27	S28	5.26	0.91	S20	0.62	1.27
S19	30	12.36	S33	4.36	2.84			
S10	26	2.21	S31	4.29	73.62			

real applications. Each of these is a real application, not a benchmark. Some are servers, and some are client applications.⁶

Table 6 provides a high level overview of footprint problems detected by the tool system. Each row of the table is a row of the output of the tool: the region in which the problem occurs, the problematic pattern, and the overhead that would be saved by fixing that pattern. The next section shows that the number of rows that user must inspect is typically low; in the worst case of the snapshots we document, the user must inspect 20 rows.

For the purposes of this section, we have selected some example pattern occurrences that would be particularly easy to fix. In a few cases, you will see that this may not cover a very large fraction of the total overhead; these are the cases where it would be necessary to fix a dozen, rather than a handful of occurrences. Still, even in these cases, fixing a few lines of code can reduce memory footprint by almost 100MB — not bad for a few minutes’ work. We now step through some of these cases in more detail.

Application S7. This application has a heap size of 652MB of which 517MB is overhead. The application suffers from three easy to fix problems in three collections. As shown in the S7 rows of Table 6, these belong to the sparse, small, and fixed-size collection patterns. One of the collections, a `HashMap` suffers simultaneously from both the sparse (P4) and the small (P3) collection patterns. The small collections are likely to be sparse, too. The tool has split out the costs of these two separate problems, so that the user can gauge the benefit of tackling these problems, one at a time: these two problems explain 92MB and 73MB of overhead, respectively.

The tool also specifies the remedies available for each pattern. For example, the small sparse `HashMap`s can be remedied by passing an appropriate number to the constructor. In addition to reporting the occurrence’s overhead (as shown in each row of Table 6), the tool (not shown) also reports the occurrence *count*, and

⁶ Their names are, unfortunately, confidential.

Table 6. Each row, a pattern occurrence, was selected as a case that would be easy for the developers to fix. As Sect. 6 shows, the developer needn’t ever fix more than 20 separate problems (and often far fewer than that) to address overhead issues.

		Total Heap Size Overhead(TO)		Region Pattern	Occurrence Overhead	Total Overhead Explained [Size] [% of TO]	
S7	652MB	517MB		HashMap P4: sparse collections	92MB	206MB	40
				HashMap P3: small collections	73MB		
				HashMap P3: small collections	19MB		
				HashMap P2: fixed-size collections	22MB		
S15	1.4GB	971MB		HashSet P1: empty collections	50MB	85MB	9
				LinkedList P1: empty collections	6.9MB		
				ArrayList P1: empty collections	6.9MB		
				sPropContainer P6: boxed scalar collections	21MB		
S3	2.61GB	2.26GB		HashMap P6: boxed scalar collections	306MB	1.1GB	49
				SparseNode P8: highly delegated	19MB		
				HashMap P6: boxed scalar collections	267MB		
				SparseNode P8: highly delegated	25MB		
				UnmodifiableMap P7: wrapped collections	108MB		
ConcurrentHashMap P1: empty collections	99MB						
S8	1.28GB	1GB		TreeMap P6: boxed scalar collections	742MB	806MB	79
				TreeMap P6: boxed scalar collections	64MB		
S32	9GB	4GB		ArrayList P4: sparse collections	736MB	861MB	21
				ObjArray P4: sparse collections	72MB		
				RedirectHashMap P4: sparse collections	34MB		
				ArrayList P3: small collections	19MB		
S27	506MB	307MB		HashSet P1: empty collections	16MB	22MB	7
				HttpRequestMI P3: small collections	6.02MB		
S17	1.872GB	1.21GB		ArrayList P4: sparse collections	53MB	84MB	7
				BigDecimal P10: sparse references	18MB		
				Date P10: sparse references	13MB		
S29	832MB	452MB		Vector P3: small collections	107MB	143MB	32
				Vector P4: sparse collections	21MB		
				Vector P1: empty collections	15MB		
S4	2.87GB	2.47GB		HashMap P9: nested collections	422MB	422MB	17

a distribution of the sizes of the collection instances that map to that pattern occurrence. This data can be helpful in choosing a solution. For example, if 90% of the `HashMap`s that map to that occurrence have only 6 entries, then this, plus a small amount of slop, is a good figure to pass to the `HashMap` constructor. For now, the tool gives these numbers, and a set of known general solutions to the pattern of each occurrence. Though this arithmetic work necessary to gauge the right solution, is straightforward, we feel that it is something the tool should do. Work is in progress to do this arithmetic automatically.

Application S3. The application uses 362MB on actual data and 2.26GB on overhead. This application suffers from several *boxed scalar collection* pattern occurrences in `HashMap`s, accounting for 573MB of overhead. There are easy to use, off the shelf solutions to this problem, including those from the Apache Commons [1] and GNU Trove [5] libraries.

The tool also finds a large occurrence of a *wrapped collections* pattern. This region is headed by a collection of type `Collections$UnmodifiableMap`; in the Java standard libraries, this is a type that wraps around a given `Map`, changing its accessibility characteristics. The tool (not shown) reveals an occurrence count of 2,030,732 that accounts for 108MB of overhead. The trivial solution to this problem is to avoid, at deployment time, the use of the unmodifiable wrapper.

Application S8. The application consumes 1.28GB, of which 1GB is overhead. As with Application S3, the main contributors to the overhead are occurrences of *boxed scalar collection* pattern. In this case, the guilty collections are two `TreeMaps`; one is especially problematic, being responsible for 742MB of overhead. Upon consultation with the developers, we learned that this application does not need the sorted property of the map until the map is fully populated. A solution that stores the map as parallel arrays of the primitive data, and sorts at the end would eliminate this overhead entirely — thus saving 1GB of memory.

Application S32. This application has a memory footprint of 9GB, of which 4GB is overhead. The main findings belong to the *sparse collection* pattern. The largest occurrence is an `ArrayList` region that consumes 1.43GB (not shown), and 736MB of these lists are used by empty array slots.

Application S4. The application spends 407MB on actual data and 2.47GB on overhead. The tool’s main finding is a *Hash Map of ArrayList* pattern which accounts for 422MB of overhead. In this case, the single outer map had many inner, but relatively small, lists. Though not small enough to fire the *small collections* pattern, this case fires the nested collections pattern. In general for this pattern, if the outer collection has a significant larger number of elements than the inner collection, the memory overhead may be reduced by switching the order of collection’s nesting. The benefit comes as a consequence of greatly reducing the total number of collection instances.

6 Validation and Characterization

The detection of costly memory patterns provides support for understanding how a particular system uses (or misuses) its memory. In this section we look at the results of our analysis across a range of real-world applications, with two goals in mind. First, we aim to validate that the approach, along with the particular patterns, produces effective results. Second, we employ the analysis to shed light on how Java programmers introduce inefficiencies in their implementations. This characterization can help the community better determine what sort of additional tools, optimizations, or language features might lead to more efficient memory usage. There are three broad questions we would like to assess: 1) Do the patterns we have identified explain a significant amount of the overhead? 2) How often do these patterns occur in applications? and 3) Do the patterns provide the best candidates for memory footprint reduction? To achieve this we

introduce a set of metrics, and apply them on a set of fifteen real-world applications. In summary, applying the metrics on the application set shows that the memory patterns we detect do indeed explain large sources of overhead in each application.

In Fig. 1 we present examples of applications with large footprint overhead. For our study, we select from these the fifteen applications which make the least efficient use of memory, applications where more than 50% of the heap is overhead. Note that in the reporting of results, we are careful to distinguish between a *pattern category*, such as *Empty Collections*, and a *pattern occurrence*, an instantiation of a pattern category at a particular context. In all of the computations of overhead explained by the tool, we consider only pattern occurrences which account for at least 1% of the application overhead. We choose this threshold so that we report and ask the user to remediate only nontrivial issues. We do not want to burden the user with lots of insignificant findings (i.e. of just a few bytes or kilobytes). To ensure meaningful comparisons, the computation of total overhead in the heap is based on traversing the entire heap and tabulating lower-level overheads like object header costs, as described in 3.4 (i.e. it is not dependent on pattern detection). Thus it includes all sources of overhead, from trivial and nontrivial cases alike.

6.1 How Much of the Overhead Do the Patterns Explain?

An important measure of the effectiveness of the analysis approach, and of the particular patterns we have identified, is whether they explain a significant portion of the memory overhead. We introduce an *overhead coverage* metric to quantify this.

Definition 4. *Overhead coverage measures the percentage of total memory overhead explained by the memory patterns detected.*

Table 1 gives a summary of the coverage across all the heaps we analyzed (not just the subset of fifteen with high overhead). In almost half of the heaps the tool is able to explain more than 80% of the overhead. Fig. 8 gives a more detailed look at the fifteen highest overhead heaps. The third bar shows us the percentage of overhead explained by 100% of the discovered pattern occurrences.

The remaining, unaccounted for overhead can be useful as well in helping us identify important missing patterns. In continuing work we are looking at the unexplained part of the heap, and are identifying which portion is the result of detecting trivial occurrences of known patterns, and which are new patterns that need to be encoded.

6.2 How Many Contexts Does a User Need to Look at?

We would like to assess if an application which contains memory inefficiencies has its problems scattered around the entire application or if the main sources of overhead are located in just a few contexts. As one validation of the usefulness

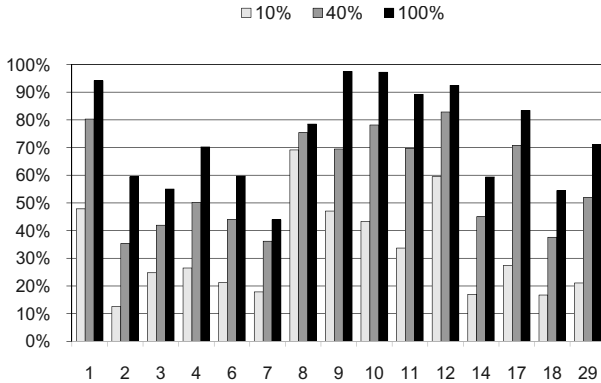
of the pattern detection tool, this can tell us whether a user can examine a manageable number of reported contexts and have a significant impact on memory footprint. From the tool perspective, the metric provides understanding into the depth of the patterns occurrences that need to be reported in order for the tool to be useful. This result is also useful from a characterization standpoint, since it can suggest whether problems are better addressed by automated optimizations (e.g. if they are diffuse) rather than by tools (e.g. if they are concentrated). The tool reports pattern occurrences ordered by the amount of overhead they explain. We would like to know how significant a part of the memory overhead is explained by the first reported findings. To achieve this we introduce the *Pattern occurrence concentration metric*.

Definition 5. *Pattern occurrence concentration shows the percentage of the heap overhead which is explained by the top $N\%$ of the memory pattern occurrences detected.*

Fig. 8 presents the results of the pattern occurrence concentration metric applied to the experimental set of applications. The figure reveals the percentage of the heap overhead which is explained by $N\%$ of the top memory pattern occurrences detected, where N is 10%, 40% and 100%. We see that on average more than 25% of the overhead is explained by the top 10% of the findings. In particular Application 8 has almost 70% of the overhead explained by the top 10% of pattern occurrences. This 70% of overhead is uncovered by one single pattern occurrence, as seen in Fig. 8(b). This means that the user has to apply only one fix to greatly reduce the application's overhead footprint. Having a single context explain the bulk of the overhead is not typical; usually there are multiple contexts that need to be addressed. From the results we can also see that increasing the percentage of pattern occurrences from 40% to 100% does not provide a significant increment in the total overhead explained, compared to the number of patterns needed to explain that difference. This means that the applications have a few important memory issues and the rest are smaller findings. For instance, in the case of Application 4 the top 40% and 100% of occurrences explain 50% and 70% of the overhead respectively. 12 pattern occurrences are required to uncover that additional 20% of overhead. Thus we can conclude that the main contributors to the memory overhead are exposed by a few top occurrences.

6.3 How Many Different Types of Problems Exist in a Single Heap?

The previous metric offers quantitative information about how much of the heap overhead is explained by the top pattern occurrences. Next, we want to identify whether an application suffers from a wide range of memory issues or if it contains only a few types of problems. Moreover, as in the previous metric, we would like to understand how many pattern categories are needed to account for a significant portion of the overhead in a given heap.



(a) The percentage of total overhead explained by top N% of pattern occurrences

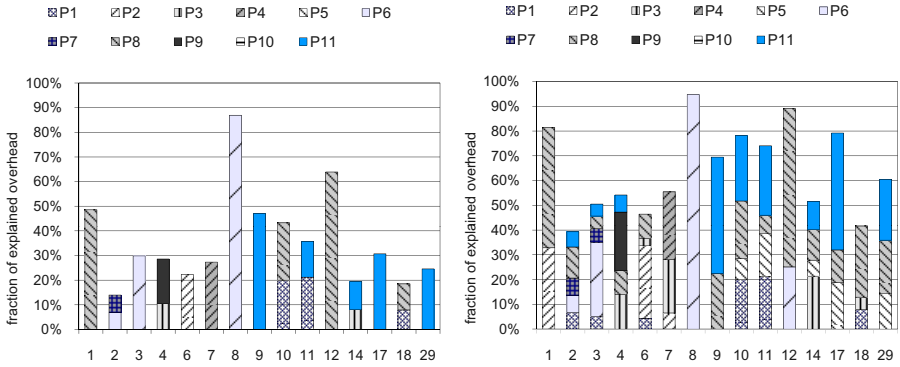
	1	2	3	4	6	7	8	9	10	11	12	14	17	18	29
10%	1	2	2	2	2	1	1	1	2	2	1	2	1	2	1
40%	2	6	6	8	8	3	2	2	5	6	2	8	4	8	4
100%	4	14	13	20	18	6	3	3	11	13	4	18	8	19	10

(b) Number of pattern occurrences in the top N%

Fig. 8. Occurrence Concentration

Definition 6. Pattern category concentration is the percentage of the heap overhead which is explained by the pattern categories represented in the top N% of memory pattern occurrences detected.

Fig. 9(a) and Fig. 9(b) depicts the category concentration results using the top 10% and 40% of findings respectively. The results show that there are always only a few categories of memory issues responsible for most of the overhead. Application 8 exhibit one type of memory inefficiency and most of the applications have two or three major contributors to the memory overhead. If we compare the pattern categories reported in both cases, for N=10% and N=40%, we note that there are no major new types of problems detected by increasing the number of pattern occurrences, even when more overhead is explained. Table 7 gives a numerical summary of the data. We can see that usually a system suffers from a small number of types of memory inefficiency, for N=40% there are 3 different issues. We can also observe that the same issue is seen in several different points in the application. This may be a consequence of the degree of reuse common in framework-based applications, or of developers coding with a similar style in multiple places in the code.



(a) The percentage of overhead explained by the pattern categories represented in the top N=10% of pattern occurrences (b) The percentage of overhead explained by the pattern categories represented in the top N=40% of pattern occurrences

Fig. 9. Category Concentration

6.4 How often Do the Pattern Categories Appear across Applications?

We have seen that the top findings reveal the main categories of memory inefficiencies usage in a given application. The next question is how often are the same patterns seen across different applications? Does a small number of pattern categories explain the bulk of the overhead in most applications, or is there more variety among applications? To study this across multiple systems we introduce a pairwise similarity metric.

Definition 7. Pattern category similarity *measures the degree to which two applications contain the same pattern categories. The similarity metric reports the ratio of the number of pattern categories common to both applications to the total number of pattern categories detected in the two applications:*

$$CS = \frac{2|PC_1 \cap PC_2|}{|PC_1| + |PC_2|}$$

where *PC* is the set of pattern categories detected in the applications 1 and 2.

The value of pattern category similarity metric belongs to [0, 1]. A value of 1 means that the same pattern categories have been detected in both applications. The lower the value of the similarity metric the greater the range of problems identified across two applications.

Fig. 10 reports the similarity metric, computed pairwise for the heaps in our experimental set. The darker the gray shade, the more common the problems detected between the two applications. To understand how a given heap compares to each of the other heaps, look at both the row and column labeled with

Table 7. Number of pattern categories represented in the top 10% and the top 40% of pattern occurrences

	# categories		
	min	median	max
top 10% of occurrences	1	1	2
top 40% of occurrences	1	3	5

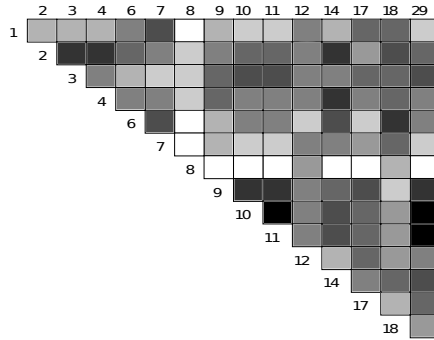


Fig. 10. Pattern category similarity. The darker the gray shade, the more common are the pattern categories found in the two applications.

the given heap (i.e. an L-shaped region of cells). There is no single application that presents completely different pattern categories compared with all the other applications, though application 8 is the least similar to the others. Eleven applications out of fifteen contain half of the categories in common with at least 9 applications (i.e. $CS \geq 0.5$). From the results we conclude that the same memory problems are frequently seen across multiple applications.

6.5 Additional Sources of Inefficiency

The current work addresses patterns of memory usage that have a high representational overhead, using a definition of overhead based on infrastructure costs such as object headers and collection implementations. Developers also introduce inefficiencies in the way they represent the data proper. For example, in our experience we have seen many applications with large amounts of duplicate immutable data, scalar data such as enumerated types that are represented in text form, or classes carrying the cost of unused fields from overly general base classes. In future work we would like to address these inefficiencies by encoding data patterns into the same framework. Much of the existing analysis approach can be easily extended to support recognition of these patterns.

7 Related Work

We discuss related work in two main categories.

Patterns. One other recent work has adopted a similar approach to memory footprint issues with a focus on collections [16]. That paper presents a solution based on a language for specifying queries of heap structure. We have found that their language, and the system as a whole, is insufficient for expressing important patterns. They hard-code collection types, and do not treat any issues outside the scope of those collections; e.g. there is no discussion of delegation or sparse references for the items stored within collections. Their aggregation is based on allocation context of *individual* objects, with, in some cases, a hard-coded “k” of context in order to cope with the implementation details of the common collection classes.

Memory Tools. Many tools put an emphasis on detecting memory leaks [12,7,17,14], rather than footprint. Several tools place an emphasis on *dominance* [3,18,10], and a great many tools [2,7,3,18,17,16] use the raw data types of the program as the basis for aggregation and reporting. Some works focus on the visualization side of memory analysis. They attempt, through cleverly designed views, to allow the human to make sense of the heap [2,6], or JVM-level behaviors [15]. These are indispensable tools for experts, but our experience shows these to be of less value among the rest of the development teams.

8 Conclusions

In Java, it is difficult to implement a data model with only a single data type. The language’s lack of support for composition and unions forces us to delegate functionality to side objects. The sometimes perverse focus on reuseability and pluggability in our frameworks [11] encourages us to favor delegation over subclassing [4,8]. For these reasons, classes are a low level manifestation of intent. In Java, even the most basic of data types, the string, requires two types of objects and delegation: a `String` pointing to a character array.

There is a wide *modeling gap* between what programmers intend to represent, and the ways that the language and runtime encourage or force them to store this information. As a consequence, most Java heaps suffer from excessive implementation overhead. We have shown that it is possible to identify a small set of semantic reasons for the majority of these overheads in Java heaps. In the future, we would like to explore this modeling gap more thoroughly. It is possible that a more rigorous study of the gap will yield opportunities close it, for important common cases. Why must we use collections explicitly, to express concerns that are so highly stylized: relationships, long-lived repositories, and transient views?

Acknowledgements. Adriana E. Chis is funded by an IRCSET/IBM Enterprise Partnership Scheme Postgraduate Research Scholarship.

References

1. Apache: Commons library, <http://commons.apache.org>
2. De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., Yang, J.: Visualizing the Execution of Java Programs. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 151–162. Springer, Heidelberg (2002)
3. Eclipse Project: Eclipse memory analyzer, <http://www.eclipse.org/mat/>
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
5. GNU: Trove library, <http://trove4j.sourceforge.net>
6. Hill, T., Noble, J., Potter, J.: Scalable visualizations of object-oriented systems with ownership trees. *J. Vis. Lang. Comput.* 13(3), 319–339 (2002)
7. Jump, M., McKinley, K.S.: Cork: dynamic memory leak detection for garbage-collected languages. In: Symposium on Principles of Programming Languages (2007)
8. Kegel, H., Steimann, F.: Systematically refactoring inheritance to delegation in a class-based object-oriented programming language. In: International Conference on Software Engineering (2008)
9. Lengauer, T., Tarjan, R.E.: A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.* 1(1), 121–141 (1979)
10. Mitchell, N., Schonberg, E., Sevitsky, G.: Making sense of large heaps. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 77–97. Springer, Heidelberg (2009)
11. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to java runtime bloat. *IEEE Software* 27, 56–63 (2010)
12. Mitchell, N., Sevitsky, G.: Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003)
13. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: Object-oriented Programming, Systems, Languages, and Applications, pp. 245–260. ACM, New York (2007)
14. Novark, G., Berger, E.D., Zorn, B.G.: Efficiently and precisely locating memory leaks and bloat. In: Programming Language Design and Implementation, pp. 397–407. ACM, New York (2009)
15. Printezis, T., Jones, R.: Gcspy: an adaptable heap visualisation framework. In: Object-Oriented Programming, Systems, Languages, and Applications, pp. 343–358. ACM, New York (2002)
16. Shacham, O., Vechev, M., Yahav, E.: Chameleon: adaptive selection of collections. In: Programming Language Design and Implementation, pp. 408–418. ACM, New York (2009)
17. Xu, G., Rountev, A.: Precise memory leak detection for java software using container profiling. In: International Conference on Software Engineering, pp. 151–160. ACM, New York (2008)
18. Yourkit LLC: Yourkit profiler, <http://www.yourkit.com>

Reuse, Recycle to De-bloat Software

Suparna Bhattacharya¹, Mangala Gowri Nanda², K. Gopinath¹,
and Manish Gupta²

¹ Indian Institute of Science

² IBM Research

{suparna,gopi}@csa.iisc.ernet.in,

{mgowri,manishgupta}@in.ibm.com

Abstract. Most Java programmers would agree that Java is a language that promotes a philosophy of “create and go forth”. By design, temporary objects are meant to be created on the heap, possibly used and then abandoned to be collected by the garbage collector. Excessive generation of temporary objects is termed “object churn” and is a form of software bloat that often leads to performance and memory problems. To mitigate this problem, many compiler optimizations aim at identifying objects that may be allocated on the stack. However, most such optimizations miss large opportunities for memory reuse when dealing with objects inside loops or when dealing with container objects.

In this paper, we describe a novel algorithm that detects bloat caused by the creation of temporary container and String objects within a loop. Our analysis determines which objects created within a loop can be reused. Then we describe a source-to-source transformation that efficiently reuses such objects. Empirical evaluation indicates that our solution can reduce upto 40% of temporary object allocations in large programs, resulting in a performance improvement that can be as high as a 20% reduction in the run time, specifically when a program has a high churn rate or when the program is memory intensive and needs to run the GC often.

1 Introduction

There are many forms of software bloat [1,2]. The creation (and deletion) of many temporary objects in Java programs is known as temporary object churn; this is the form of software bloat that we address in this paper. As illustrated by Jack Shirazi [3], creating too many temporary objects results in higher garbage collection overhead, object construction costs and higher memory system stress resulting in an increase in processing time and memory consumption. At the end of the chapter on object creation in his book, Shirazi gives a long list of performance improvement strategies of which we reproduce a few here:

- Reduce the number of temporary objects being used, especially in loops.
- Avoid creating temporary objects within frequently called methods.

```

class Klass {
    Hashtable ftab;
    void foo(int num, Hashtable tab) {
5     HashSet seen = new HashSet();
6     Stack work = new Stack();
7     Vector heap = new Vector();
8     doSomething(work, num);
9     while ( !work.isEmpty() ) {
10      Object w = work.pop();
11      if ( seen.contains(w) )
            continue;
12      seen.add(w);
13      heap.add(w);
        }
        Integer inum = new Integer(num);
14      if ( init() ) {
15          ftab.put(inum, heap);
        }
        else {
16          tab.put(inum, heap);
        }
    }

    void bar(int num) {
32     Hashtable tab = new Hashtable();
33     for ( int n=0; n<num; n+=10 ) {
34         foo(n, tab);
35     }
43     dumpTabContent(tab);
    }

    void driver() {
44     for ( int num=0; num<100; num+=5 ) {
45         bar(num);
        }
    }
}

```

(a) sample code

```

class Klass {
    Hashtable ftab;
    void foo(int num, Hashtable tab) {
5     HashSet seen = REUSE.REUSE_01();
6     Stack work = REUSE.REUSE_02();
7     Vector heap = new Vector();
8     doSomething(work, num);
9     while ( !work.isEmpty() ) {
10      Object w = work.pop();
11      if ( seen.contains(w) )
            continue;
12      seen.add(w);
13      heap.add(w);
        }
        Integer inum = new Integer(num);
14      if ( init() ) {
15          ftab.put(inum, heap);
        }
        else {
16          tab.put(inum, heap);
        }
    }

    void bar(int num) {
32     Hashtable tab = new Hashtable();
33     for ( int n=0; n<num; n+=10 ) {
34         foo(n, tab);
35     }
43     dumpTabContent(tab);
    }

    class REUSE {
52     static HashSet hs_01 = new HashSet();
53     HashSet REUSE_01() {
54         hs_01.clear(); return hs_01;
        }
57     static Stack st_02 = new Stack();
58     Stack REUSE_02() {
59         st_02.clear(); return st_02;
        }
    }
}

```

(b) Code reused

Fig. 1. Sample code

- Reuse objects where possible.
- Empty collection objects before reusing them. (Do not shrink them unless they are very large.)

However, this is easier said than done, especially for Java programmers who have grown up with the luxury of creating and discarding temporary objects, on the assumption that the discards would be efficiently garbage collected. Consider, for example, a typical piece of Java code as shown in Figure [1\(a\)](#):

In this program, `foo()` calls `doSomething()` which loads several objects into a stack `work`. Then `foo()` picks up each element in the stack, checks for and discards any duplicates using `seen` and loads the unique objects into `heap`. At the end, based on some condition, `heap` is stored into either the field variable `this.ftab`

or into the `Hashtable` `tab`¹ passed in as a parameter². The method `bar()` calls `foo()` iteratively and then dumps the contents of `tab` while the method `driver()` calls `bar()` iteratively.

Here we observe that `foo()` is called from inside a loop. Hence `HashSet` `seen`, `Stack` `work` and `Vector` `heap` will be created once for every iteration of the loop. Also, it is intuitively clear that `seen` and `work` can be reused, but `heap` may not be reusable.

- Consider `Stack work`: it is created locally and is not accessible outside `foo()`—that is, it does not escape `foo()`. It may be reused as shown in Figure 1(b). Note, however, that the enclosing loop is in a different method than the objects being reused and thus requires interprocedural analysis. Nevertheless, `work` is reusable within the innermost enclosing loop and hence is termed a “Level 1” reusable object.
- Consider `Vector heap`: it is created locally but it is accessible outside `foo()`—that is, it escapes `foo()`.
 - Consider the case when it escapes via `tab`: `heap` does not escape the method `bar`, but it does “escape” the loop inside `bar`. Going further back up the call flow graph, we find that `bar` is called from within a loop. Since `heap` does not escape from this loop, it is potentially reusable. In this case, `heap` is not reusable within the innermost enclosing loop, but it is reusable within the next enclosing loop and hence is termed a “Level 2” reusable object.
 - When it escapes via `ftab`: `ftab` is accessible outside `bar` and `driver` and hence so is `heap`. Therefore, `heap` is not reusable along this path.

When we reused `seen` and `work` as shown in Figure 1(b), we observed a 9% reduction in execution time (on a dual core Intel(R) Core(TM)2 Duo system with 2GB RAM running Java Hotspot(TM) Server VM on Linux).

Thus we see that objects may be reused within the immediately enclosing loop or a higher level loop. The same object may be reusable along one path but not another. Similarly, the same object may be reusable at different levels along different paths. Besides these, there are many issues related to this kind of code transformation:

1. How do we determine automatically which variable can be reused and which one cannot be
2. Which data structures do we target and how do we know how to “clear” the structure before reuse
3. How do we determine when to perform the allocation and the “clear”. In the example, we have given a trivial solution which does not always work
4. Where do we insert the reuse code so that it does not become an overhead in itself

¹ For ease of exposition we model a hashtable as directly containing the key and value fields *e.g.*, `tab.value` instead of containing the fields only indirectly *e.g.*, `tab.bucket[i].element[j].value`.

² This code was modified from Xylem code (refer Section 5), the only modification being the addition of `ftab` and the corresponding lines of code at lines 14 and 15 to highlight that an object may be reusable along one path but not another.

Although, in principle, it is possible to reuse any data structure, in our implementation, we address only certain Collection classes—specifically `HashSet`, `Vector`, `Stack`, `PriorityQueue`, `LinkedList`, `ArrayList` and `TreeMap`. This makes it easy to clear the objects using the `clear()` method from the Collection class. We also reuse memory in Strings (here we are referring to the reuse of the underlying arrays and not reuse of the string representation by string interning). This is far more complex and the details are given in Section 4.

Contributions. In this paper we give a novel algorithm for automatically finding sources of software bloat and then we give a solution to transform the code to reduce the bloat. The main contributions are:

- An algorithm that can detect objects created within a loop and determine whether an object created within a loop can be reused at the end of each iteration. In the case of nested loops, the algorithm will tell us the innermost enclosing loop in which the object can be reused.
- A solution that can automatically transform the source code to reuse the object such as to mitigate the effects of software bloat.
- An implementation that validates our claims and shows that we can get upto 40% reduction in bytes of temporary objects generated and 20% improvement in speed of execution.

Organization. We start off with some definitions and a description of escape analysis used in this paper in Section 2. In Section 3 we explain how to find safe reusable allocations and in Section 4 we give an algorithm that achieves the reuse through a source-to-source transformation. In Section 5 we report the empirical justification for using our analysis, Section 6 positions our work with respect to related work and we conclude in Section 7.

2 Preliminaries

The *control-flow graph* (CFG) for a method M contains *nodes* that represent statements in M and *edges* that represent potential flow of control among the statements.

We define here some terms used in the paper.

Definition 1. Dominator: A node S_i dominates a node S_j iff $S_i \neq S_j$ and S_i is on every path from Entry to S_j .

Definition 2. Postdominator: A node S_j postdominates a node S_i iff $S_i \neq S_j$ and S_j is on every path from S_i to Exit.

Definition 3. Control dependence: A node S_j postdominates a branch of a predicate S_i iff S_j is the successor of S_i in that branch or S_j postdominates the successor of S_i in that branch.

A node S_j is control dependent on a predicate S_i iff S_j postdominates a branch of S_i but S_j does not postdominate S_i . A node can be directly control dependent on itself. Note that a node with only one successor can never be the source of a control dependence edge.

Definition 4. *Backedge:* A backedge in the CFG is an edge where the destination of the edge dominates the source of the edge.

Definition 5. *Data Dependence:* A node S_j is data dependent on a node S_i , if S_i defines some variable x , S_j uses the variable x , and there exists a path from S_i to S_j without intervening definitions of x .

Definition 6. *Loop Carried Data Dependence:* A node S_j is data dependent on a node S_i , if S_i defines some variable x , S_j uses the variable x , and there exists a path from S_i to S_j without intervening definitions of x and the path contains a backedge.

Escape Analysis. To locate reuse possibilities, we use escape analysis which is a method for determining the dynamic scope of pointers. After constructing the control-flow graph of each method, our solution uses flow- and context-sensitive pointer analysis and escape analysis³. The escape analysis computes the escape-in and escape-out sets for each method.

- The *formal-in* set for a method M contains the set of formal parameters. The implicit *this* parameter (in non-static methods) is also a formal-in.
- The *formal-out* set for a non-void method M contains a single parameter R , the designated return value. The *formal-out* set is empty for a void method.
- The *escape-in* set for a method M contains direct and indirect fields of the formal parameters of M that are used, before possibly being defined, in M . These represent the upwards-exposed uses in M .
- The *escape-out* set for M contains direct or indirect fields of the formal parameters of M and the return value of M that are defined in M .
- At each *Call* site c that calls method M , the algorithm uses the escape-in and escape-out information, to compute the actual-in and actual-out sets, where
- we generate an *actual-in* for each formal-in and each escape-in and
- we generate an *actual-out* for each formal-out and escape-out in M .

The algorithm associates escape-in and formal-in sets with the *Entry* node of the CFG, and escape-out and formal-out sets with the *Exit* node of the CFG; likewise, the actual-in and actual-out sets are associated with call sites.

In the example shown in Figure 2, `num` (node 2) and `tab` (node 3) are formal-parameters in the method `foo` as is the `this` parameter although it is not shown explicitly in the figure. `this.ftab` (node 4) is an escape-in parameter where `ftab` is a field of the formal-in parameter `this`. There is no formal-out parameter as both the functions are void functions, but `this.ftab.key` (node 20) and `this.ftab.value` (node 21) are escape-out parameters generated from

³ A *context-sensitive* analysis propagates states along interprocedural paths that consist of valid call–return sequences only—the path contains no pair of call and return that denotes control returning from a method to a call site other than the one that invoked it. A *flow-sensitive* analysis, on the other hand, takes into account the order of statements in a program.

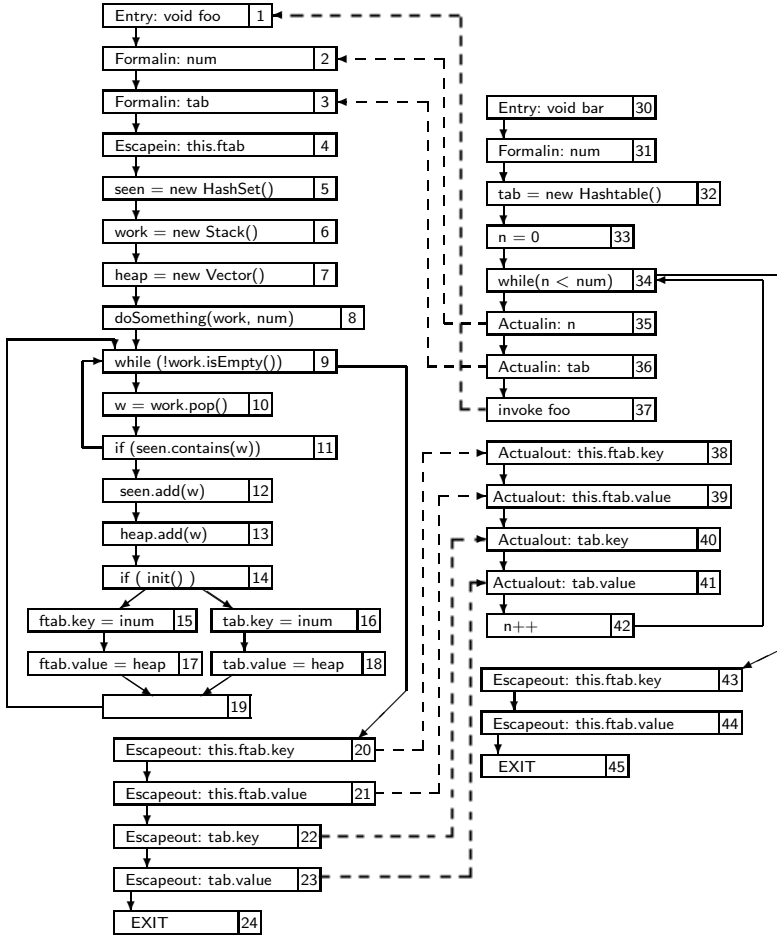


Fig. 2. Escape Analysis

the `put` method of the `Hashtable`. Similarly, `tab.key` (node 22) and `tab.value` (node 23) are escape-out parameters since they are fields of the formal-in `tab`.

In method `bar` at the call site for `foo` (node 37) we have generated actual-ins and actual-outs and mapped them appropriately to the formal-in and escape-out parameters in the called function.

3 Finding Potential Sources of Bloat

In this section, we describe how to locate object allocation sites within loops that can be reused. Our analysis first identifies whether an allocation site is within a loop. This allocation site can be converted into a reuse site on the condition that

1. it does not have a loop carried dependence
2. it is not accessed outside the loop. To determine whether it is local to an enclosing loop, we need to check if it escapes the scope of the loop.

3.1 The Problem with Loop Carried Data Dependence

Consider the following piece of Java code

```
Vector vprev = new Vector();
while ( cond ) {
    vsucc = new Vector();
    process(vsucc);
    if ( vsucc.size() <= vprev.size() ) {
    }
    vprev = vsucc;
}
```

Here there is a loop carried dependence from vsucc to vprev. So if we reset and reuse vsucc inside the loop, then after the first iteration vsucc and vprev will always point to the same Vector, which is not correct.

Knowing that vsucc can be reused after N cycles, it is possible to design reuse as follows:

```
Vector tmp[] = new Vector()[N];
for (int j=0; j<N; j++) {
    tmp[j] = new Vector();
}
Vector vprev = new Vector();
int i=0;
while ( cond ) {
    tmp[i].clear();
    vsucc = tmp[i];
    i++;
    if ( i == N ) {
        i = 0;
    }
    process(vsucc);
    if ( vsucc.size() <= vprev.size() ) {
    }
    vprev = vsucc;
}
```

Finding loop carried dependence is relatively simple. However, it is not always possible to determine statically after how many cycles vsucc would be reusable, as shown in the example below.

```
Vector vprev = new Vector();
while ( cond ) {
    vsucc = new Vector();
    process(vsucc);
    if ( vsucc.size() >= vprev.size() ) {
```

```

    vprev = vsucc;
    ...
}
}

```

Hence, we conservatively ignore reuse when there is a loop carried data dependence.

3.2 The Basic Algorithm

The preliminary analysis consists of the following steps

- We compute flow and context sensitive data and control dependence, wherein the data dependence includes points-to and escape analysis described in the previous section. Each data dependence that is a loop carried dependence is flagged appropriately.
- We determine which conditionals in the Java bytecode are loop conditionals. We define a loop header as the destination of an edge in the CFG such that the node at the destination of the edge dominates the node at the source of the edge.
- We find all allocation sites in the code. For each allocation site S_{new} we compute the transitive closure of control dependencies. This process is performed interprocedurally. If S_{new} is not directly or transitively control dependent on a loop header, then we can discard it as being uninteresting from the point of reuse.

Computing transitive closure of control dependencies. Intra-procedurally speaking, every node is eventually control dependent on the ENTRY node of the method. The ENTRY node is inter-procedurally control dependent on the CALL node from where the method is invoked. The transitive closure thus includes all nodes that the CALL node is control dependent upon.

In Figure 2, `HashSet seen = new HashSet()` is control dependent on the ENTRY node `void foo()`. The ENTRY node is inter-procedurally control dependent on the CALL node `foo(n, tab)` in the method `bar()`. The CALL node is control dependent on the for conditional `n < num`. Hence, the allocation `seen = new HashSet()` is inter-procedurally and transitively control dependent on a loop header.

Removing Unnecessary Loop Header Dependencies. Note that a node that is control dependent on a loop header is not necessarily within the loop. However, we are only interested in finding allocations that are within a loop and hence need to perform additional computation.

All nodes within the loop are directly or transitively control dependent on the loop header. However there may be nodes outside the loop that are also control dependent on the loop header. This happens when there is a return from within the loop or when there is an exception flow edge from within the loop. Since we are interested only in nodes within a loop, we need to filter out these

Algorithm 1. Locating reusable allocations within a loop

```

1: INPUT:  $S_{new}$ 
2:  $\phi_{esc} \leftarrow$  new Collection()
3:  $\phi_{reg} \leftarrow$  new Collection()
4: computeForwardSlice( $\{S_{new}\}, \phi_{esc}, \phi_{reg}$ )
5:  $N_{CD} \leftarrow$  controlDepPred( $S_{new}$ )
6: while  $N_{CD} \neq$  null do
7:   if  $N_{CD}$  is a loop header then
8:     level++
9:     if  $\phi_{esc} = \{\}$  and contains( $N_{CD}, \phi_{reg}$ ) and noLoopDD( $\phi_{reg}$ ) then
10:      OUTPUT(level,  $S_{new}$ )
11:     end if
12:   else if  $N_{CD}$  is an Entry node then
13:     for all  $N_{invoke}$  a call site of Entry do
14:       newset  $\leftarrow$  map( $N_{invoke}, \phi_{esc}$ )
15:        $\phi_{esc} \leftarrow$  new Collection()
16:        $\phi_{reg} \leftarrow$  new Collection()
17:       computeForwardSlice(newset,  $\phi_{esc}, \phi_{reg}$ )
18:     end for
19:   else
20:     reached the top of the call graph
21:     report and exit
22:   end if
23:    $N_{CD} \leftarrow$  controlDepPred( $N_{CD}$ )
24: end while
25:
26: computeForwardSlice(newset,  $\phi_{esc}, \phi_{reg}$ ) {
27: while !newset.empty() do
28:    $N \leftarrow$  newset.removeLast()
29:   for all  $N_{dd}$  such that  $N_{dd}$  is data dependent on  $N$  do
30:     if  $N_{dd}$  is a formal-out or an escape-out then
31:        $\phi_{esc} \leftarrow \phi_{esc} \cup N_{dd}$ 
32:     else
33:        $\phi_{reg} \leftarrow \phi_{reg} \cup N_{dd}$ 
34:     end if
35:   end for
36: end while
37: }
```

external-to-the-loop nodes. We do this by simply checking if there is a path from the node to the loop header that ends with a back edge.

Having found an allocation site that lies within a loop, we perform the algorithm given in Figure 1.

The algorithm takes as input S_{new} , the allocation site $v = \text{new Collection}()$, where `Collection` is one of the classes mentioned in Section 1. It then computes the forward slice for S_{new} as explained at lines 27–36. The forward slice consists of the transitive closure of all def-use sets starting with the definition at S_{new} .

The nodes in the slice are separated into two bags, one called the “Escape” bag ϕ_{esc} that contains any **formal-outs** or **escape-outs** in the slice and the other bag ϕ_{reg} that contains all other nodes.

Next we track backward along the control dependence edges.

- If we come to a loop conditional we check if ϕ_{esc} is empty and every node in ϕ_{reg} lies within the loop and does not have a loop carried dependence. If yes, then we have found the closest enclosing loop inside which S_{new} can be reused—along this particular path. We record this path and stop traversing the control flow graph any further for this path. For all other loop conditionals or branch conditionals, continue climbing up the control dependence graph.
- If we come to the **Entry** node of a method S_M , then for each invocation site, S_{call} , we map each node in the ϕ_{esc} set to the corresponding **actual-out** nodes. The old ϕ_{esc} and ϕ_{reg} sets are discarded and fresh sets are computed as the union of the forward slices of the **actual-out** nodes at the given invocation site. Then the analysis continues up the control dependence graph,
- If we come to the top of the call flow graph, we conclude that S_{new} may not be reusable along this path.

An illustrative example. Consider the example in Figure 2.

1. We determine that the conditional nodes `n < num` (node 34) in `bar` and `!work.isEmpty()` (node 9) in `foo` are loop headers.
2. Consider the statement `seen = new HashSet()` (node 5) in method `foo` in Figure 1. It is an allocation site for the Collection class `HashSet`. This node is control dependent only on the entry node `void foo` (node 1). This node is call dependent on the `invoke foo` node (node 37) which in turn is control dependent on the loop header `n < num`. Hence the allocation statement is interprocedurally called from inside a loop and has potential to be reused.
3. The forward slice is computed as $\phi_{reg} = \{ \text{seen.contains(w)}, \text{seen.add(w)} \}$ and ϕ_{esc} is empty as none of the nodes are formal-outs or escape-outs.
4. Now we traverse the control dependence path. At the entry node there is nothing to be mapped to the `invoke foo` site as ϕ_{esc} is empty. We discard the current ϕ sets and enter the method `bar` with empty sets. Next the `invoke foo` is control dependent on the loop header `n < num`. Here the requisite conditions are trivially true. Hence the allocation may be converted to reuse.

If we consider the allocation site `heap = new Vector()`, it has four escape-outs in its ϕ_{esc} ; these map into **actual-out** nodes in the calling function `bar`. These **actual-out** statements are inside the loop but their forward slice contains nodes that are outside the scope of the loop. Two of these escape out of `bar` as well as its caller `Driver()`. Hence, this node is correctly not marked for reuse.

3.3 Multiple Control Dependence Paths

The basic analysis algorithm described above records the closest enclosing loop along a control dependence path where reuse may be implemented safely, if at

all. Since there may be multiple paths to an allocation site, several situations may arise:

1. The site is not reusable along any control dependence path
2. The site is reusable along some control dependence paths, but not reusable along other control dependence paths.
3. The site is reusable along all its control dependence paths, but the closest enclosing loop where reuse can be implemented is not the same for all control dependence paths
4. The site is reusable inside the same closest enclosing loop along all its control dependence paths

One could take a conservative approach where only Case 4 is assumed to be safe for reuse conversion. However, this tends to miss several sites with potentially large churn (as we observe experimentally). A second approach is to introduce extra code to perform runtime tracking of the conditions for safe reuse in all situations. While this can enable more opportunities for reuse, it can become fairly complicated and invasive. For example, in the worst case, this might involve interprocedurally tracking path history along every branch leading to an allocation site from enclosing loops located several call levels away.

Instead, we use a simpler scheme that achieves greater precision than the conservative analysis but only exploits runtime state that needs to be introduced anyway for implementing object reuse.

Let us define the *height* h of a loop L along a control dependence path from an allocation site as the number of enclosing loop headers along that path upto and including L . Then, the reuse level k for an allocation site along a particular control dependence path is defined as the height of the closest enclosing loop where the site is reusable for that path. This means that object reuse state for that allocation site must be maintained across iterations of all the inner loops upto height $k - 1$, and can only be reset across iterations of the loop at height k or above. As long as this condition can be met across all control dependence paths for the site without conflict, the object can be safely converted for reuse along certain paths (where it is found to be reusable) without affecting correctness along its other control dependence paths. This logic can be extended to address not just Case 3, but Case 2 as well, since a path that does not support reuse can be treated as a path with a very high reuse level. In other words, at some outer loop level we can setup one control flow path to reuse and the another to not reuse, provided the two paths do not intersect within the same outer loop iteration.

Illustration: Consider the following variation of example in Figure 2, without lines 14-15, so that the allocation to `heap` no longer escapes directly via `ftab`. Now, suppose we add a couple of routines as follows:

```
void barPersist(int num) {
    for (int n=0; n<num; n+=10) {
        foo(n, ftab);
    }
}
```

```

void driverPersist() {
    for (int num=0; num<100; num+=5) {
        barPersist(num);
    }
}
void mainDriver() {
    if (init()) {
        driverPersist()
    } else {
        driver();
    }
}
}

```

The allocation site `heap = new Vector()` in `foo()` is reusable at level $k = 2$, the loop in `driver()` that calls `bar()`, along one control dependence path, but is not reusable along another path that goes through `driverPersist()` and `barPersist()`. In this case, we notice that there is no conflict between these two cases as the corresponding loops do not intersect.

Now, let us say the routines `driverPersist()` and `mainDriver()` were removed and instead, the routine `driver()` modified as follows:

```

void driver() {
    for (int num=0; num<100; num+=5) {
        if (init()) {
            barPersist(num);
        } else {
            bar(num);
        }
    }
}
}

```

This time, the outer loop is common to the two paths, which indicates a potential conflict.

We perform an analysis of the loop sharing structure across control dependence paths to eliminate such potential conflicts.

Figure 3 illustrates some examples of loop header sharing across control dependence paths starting from two distinct nodes P and Q respectively. The loop header nodes are numbered according to the height of the loop with respect to the allocation site. In (a) the loops are embedded along both paths, hence there is a conflict if a reuse site in the the inner loop is not reusable along either P or Q. In (b), the loops are disjoint and both inner loops invoke the method containing the reuse site. In this case, for reuse level $k = 2$ and above, the the $k - 1$ loop of one path never falls inside the other. Hence if the site is reusable at level 2 (or higher) along paths from P, then, even if it is not reusable from Q, our transformation can be set up to safely exploit object reuse along the former. In (c), the innermost loop is shared, but the outer loops are disjoint. Thus a reuse transformation upto level 2 would be unsafe unless both paths share the same reuse level. However, if the site is reusable at level 3 or higher along one path, then, even if it isn't reusable along the other, our transformation can be set up safely to exploit object reuse at the appropriate level along that path.

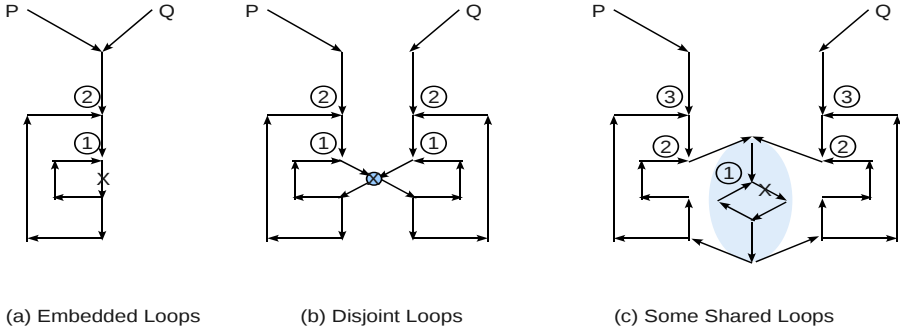


Fig. 3. Loop header sharing for multiple control dependence paths

4 Object Reuse, Recycle Transformations

In the previous section we described an approach for finding allocation sites that are candidates for object reuse and the closest enclosing loops where they can be safely reused. Now we discuss our automated code transformations for implementing object reuse.

Object reuse optimizations may involve object memory reuse or object content reuse. The former recycles the memory and structural representation state of objects of the same type, instead of allocating fresh objects each time. The latter reuses at least some part of the actual object content (a form of memoization/caching) to save repeated content construction costs.⁴ Our static analysis based detection technique mainly identifies the first kind of opportunities, hence this forms the focus of our implementation. Our code transformations can, however, be used to support the second category of reuse as well, with slight modifications.⁵

4.1 Basic Reuse-Recycle Algorithm

The static analysis phase reports the safe reuse level and the corresponding loops for each allocation site identified for reuse (hereafter referred to briefly as a reuse site). We use this information to implement a basic reuse/recycle algorithm for these reuse sites.

The simple reuse transformation illustrated in the introduction is efficient but does not work in many situations. The allocation has been moved to a static initializer where constructor parameters that are specified at the allocation site may not be available. The conversion is only applicable for level 1 reuse, i.e. for objects allocated in an inner loop which can be recycled at the next

⁴ Object canonicalization is an extreme example of content reuse; object pooling involves memory and sometimes partial content reuse.

⁵ e.g. steps like clearing the object or simulating effects of a constructor may be skipped when reusing object content, thus simplifying the implementation.

iteration of this loop. In this case, it suffices to allocate a single reusable slot for an allocation site, e.g. the variable `REUSE.st_01` for `Stack work`. However, when allocated objects need to be preserved across iterations of an inner loop, and can only be recycled at a subsequent iteration of an outer loop, multiple reusable slots must be maintained for the same allocation site. This happens for a level k reuse with $k > 1$, (where k is the height of the closest enclosing loop at which the object can be reused), e.g. $k = 2$ for `Vector heap` when `foo()` is invoked via `bar()`. In this case, static initialization cannot be used as the number of distinct slots required (minimum outstanding allocations) may not be known until the inner loop completes its first iteration sequence. It could even change dynamically. The number of inner loop iterations and hence reuse slots maintained for `Vector heap` varies with the loop upper bound `num`, e.g. when `num = 50`, 10 reuse slots are used. We note that this also means that the number of reuse slots created must be bounded to avoid causing memory overhead due to a blowup in the number of inner loop iterations.

Therefore in our generalized implementation (Algorithm 2), the allocation statement is not moved, but instead, tracked during the first iteration of the level k loop by creating reuse slots as needed and initializing them with the result of the allocations in inner loop iterations (lines 9-12). Note that `Reusevar.numslots` is statically initialized to zero. It is incremented (line 10) each time a reuse slot is created for this site, using `Reusevar.addslot()` (line 11). The objects from these slots are then reused sequentially (lines 16-17) during subsequent iterations of the level k loop. If the inner loop iterations exceed the number of available reuse slots `Reusevar.numslots` (e.g. due to a varying loop bound), then additional slots are created as required (upto a maximum allowed capacity) (lines 9-12). If the maximum capacity of reuse slots is exceeded for a given allocation site, then allocations required beyond the capacity simply fall back to a non-reusable mode (lines 13-14).

This approach has the downside of an extra check in every inner loop iteration to distinguish the first iteration⁶ of the level k loop from iterations which reuse previous allocations. The overhead may be optimized using loop peeling and specialization for common scenarios (like level 1 reuse for collection objects which do not require a constructor parameter).

To enable an existing object to be recycled instead of issuing a fresh allocation, some type specific steps need to be executed to re-initialize the object for reuse. In general, this may require simulating (a part of) its constructor functionality. We focus on reusing collection objects and strings.

4.2 Reusing Collections

Preparing a collection object for reuse is particularly simple. Most collections provide a `clear()` method to reset a collection to zero entries while keeping the capacity of the collection intact. The larger the collection being reused, the greater the benefit as it saves a large portion of object construction costs.

⁶ And checks for dynamic expansion of slots.

Algorithm 2. General level K reuse transformation

```

1: while condition  $K$  do
2:   processing for  $K^{th}$  loop
3:    $slot \leftarrow 0$ ;  $maxslots \leftarrow MAXSLOTS$  {ADDED TO ENABLE REUSE}
4:   while condition  $K - 1$  do
5:     processing loops for  $K - 2$  to 2
6:     while condition 1 do
7:       processing for inner loop
8:       BEGIN: Transformed allocation statement {TO ENABLE REUSE}
9:       if  $Reusevar.numslots \leq slot < maxslots$  then
10:         $Reusevar.numslots \leftarrow Reusevar.numslots + 1$ 
11:         $Reusevar.addslot() \leftarrow new\ TYPE(params)$ 
12:       end if
13:       if  $Reusevar.numslots \leq slot$  then
14:         $var \leftarrow new\ TYPE(params)$ 
15:       else
16:         $var \leftarrow Reusevar.getslot(slot)$ 
17:         $slot \leftarrow slot + 1$ 
18:       end if
19:       END : transformed allocation statement {TO ENABLE REUSE}
20:       some more processing for inner loop
21:     end while
22:     some more processing for loops for  $K - 2$  to 2
23:   end while
24:   some more processing for  $K^{th}$  loop
25: end while

```

4.3 Reusing Strings

Recycling String objects requires simulating a part of its constructor functionality to re-populate the underlying character array with new content. Since a String object is an immutable data structure, this can be implemented efficiently only with special extension support from the class library or the JVM. For our experimental evaluation, we use reflection to access/clear/overwrite the array as required. This incurs a performance penalty, which is mitigated to some extent by caching the reflection results when the object is first allocated to avoid the overhead on every iteration. Therefore our results provide a conservative estimate of performance gain that can be attained through object reuse in this case.

4.4 Implementation Details

We used a source to source transformation approach to evaluate the feasibility of our automated object reuse/recycle conversion. Simplicity and clarity were our primary motivation for choosing this approach, e.g. ability to perform a visual inspection of the changes in source code after transformation. The conversion may also be implemented using byte-code manipulation and JVM level optimizations as discussed later.

The inputs required for the transformation are the output of the static analysis stage, and the source files of the application to convert.

Figure 4 illustrates our running example before and after automatic reuse conversion. The transformations are performed in a single-pass over source. This is a slightly modified version of our running example from previous sections, without the `ftab` field. As `heap` no longer escapes via `ftab`, it is now reusable at level 2 (i.e. the loop starting at lineno 38 in `driver()`).

<pre> class Klass { void foo(int num, Hashtable tab) { 14: HashSet seen = new HashSet(); 15: Stack work = new Stack(); 16: Vector heap = new Vector(); doSomething(work, num); while (!work.isEmpty()) { Object w = work.pop(); if (seen.contains(w)) continue; seen.add(w); heap.add(w); } Integer inum = new Integer(num); tab.put(inum, heap); } void bar(int num) { Hashtable tab = new Hashtable(); 31: for (int n=0; n<num; n+=10) { foo(n, tab); } dumpTabContent(tab); } void driver() { 38: for (int num=100; num > 0; num-=5) { bar(num); } } </pre> <p style="text-align: center;">(a) Before transformation</p>	<pre> class Klass { void foo(int num, Hashtable tab) { 14: HashSet seen = REUSE.ReuseHashSet_14(); 15: Stack work = REUSE.ReuseStack_15(); 16: Vector heap = REUSE.ReuseVector_16(); doSomething(work, num); while (!work.isEmpty()) { Object w = work.pop(); if (seen.contains(w)) continue; seen.add(w); heap.add(w); } Integer inum = new Integer(num); tab.put(inum, heap); } void bar(int num) { Hashtable tab = new Hashtable(); 31: REUSE.idxVector_16 = 0; REUSE.maxVector_16 = MAX_SLOTS; for (int n=0; n<num; n+=10) { foo(n, tab); } dumpTabContent(tab); } void driver() { 38: for (int num=100; num > 0; num-=5) { bar(num); } } </pre> <p style="text-align: center;">(b) After transformation</p>
--	---

At each listed allocation site to convert for reuse (lines 14,15,16), we replace the call to `new` with a call to an allocation site specific reuse method which performs allocation tracking and reuse. At the statement preceding a listed allocation site's level $k - 1$ loop header, we insert code to reset the reuse slot index for the allocation site and specify the maximum slots that may be created. Line 31 is the level 1 loop header corresponding to the level 2 reusable allocation of `heap` (`idxVector_16` is the corresponding reuse slot index).

A reuse context area and allocation site specific reuse methods are generated by the transformation. The reuse context fields maintain state corresponding to every allocation site that is converted for reuse. `Stack_15` maintains a reference to the level 1 reusable allocation for `work`. The reuse methods encap-

```

public class REUSE{
  static HashSet HashSet_14;
  public static HashSet ReuseHashSet_14() {
    if (HashSet_14 != null) {
      REUSEUtil.clearHashSet(HashSet_14);
    } else {
      HashSet_14 = new HashSet();
    }
    return HashSet_14;
  }

  static Stack Stack_15;
  public static Stack ReuseStack_15() {
    if (Stack_15 != null) {
      REUSEUtil.clearStack(Stack_15);
    } else {
      Stack_15 = new Stack();
    }
    return Stack_15;
  }
  ...
}

static int idxVector_16;
static ArrayList<Vector> Slot_Vector_16 =
  new ArrayList<Vector>();
static Vector Vector_16;
public static Vector ReuseVector_16() {
  if (idxVector_16 < Slot_Vector_16.size()) {
    Vector_16 =
      Slot_Vector_16.get(idxVector_16++);
    REUSEUtil.clearVector(Vector_16);
  } else {
    Vector_16 = new Vector();
    if (idxVector_16 < maxVector_16) {
      Slot_Vector_16.add(Vector_16);
      idxVector_16++;
    }
  }
  return Vector_16;
}

```

Fig. 4. Code transformation example

sulate allocation tracking and reuse logic specific to these allocation sites, e.g. `ReuseVector_16()` uses `Slot_Vector_16` to keep track of the reuse slots for the allocation of `heap` at line 16. The size of the arraylist `Slot_Vector_16` thus corresponds to `Reusevar.numslots` in Algorithm 2. For level 1 reuse, as in the case of `seen` and `work`, there is a single reuse slot which is accessed directly from `HashSet_14` and `Stack_15` respectively. Before returning the reusable reference, these methods invoke a type-specific utility method to enable reuse for that object (`clearHashSet()` for `seen`, `clearStack` for `work` and `clearVector` for `heap`).

Reuse context entries are typically stored in a thread local reuse context area. For single-threaded programs like the above example, we maintain a global reuse context, to avoid the overhead of thread local context accesses in the interprocedural case.

4.5 Dynamic Analysis Guided Filtering of Candidate Reuse Sites

A purely static analysis based detection scheme has insufficient information to prioritize allocation sites to convert based on an estimate of expected savings. We complement it with a dynamic analysis that profiles allocation sites with high object churn to guide the selection of statically identified candidate reuse sites that are worth converting. We then apply our object reuse transformation for those reuse sites.

The dynamic analysis phase takes as input the reuse sites reported by static analysis and the output of an allocation profiler that captures the volume of allocated and live vs freed bytes generated at each allocation site under a typical run of the program. It then generates statistics about the proportion of churn generated by reuse sites which use collections or strings, and selects the top sites with significant contribution to overall volume of temporary objects generated.

4.6 Discussion

Alternatives to full source to source transformation. Instead of using a pure source to source transformation approach, object reuse transformations could also be implemented using byte code manipulation and JVM level optimizations. A JVM can avoid costs of reflection and thread local accesses that we incur and optimize the overhead of the check required in each iteration to distinguish first time allocation and reuse iterations. It can also enable profile guided object reuse conversion to be applied at runtime for the reusable allocation sites that exhibit the potential for highest savings.

Extending the technique to non-collection objects. The technique may be generalized further to any object type that is designed to support a special reuse interface with a type specific reuse method. This method provides an alternative to the constructor that is called to clear a previous instance of the object or re-populate it with new content. Such an approach can also be used to enable partial content reuse by implementing the reuse method to selectively preserve the content of some fields of the object.

5 Empirical Evaluation

We apply our analysis to a few large applications, the SPECjbb2005 benchmark and the DaCapo benchmarks [4] lusearch, ps, pmd, antlr. We also apply it to Xylem [5], a proprietary tool that has been built to statically detect null dereferences in Java. In this paper we analyze only a subset of Xylem. Table 1 lists a brief description of the benchmarks used. The freed memory was measured from the garbage collection (GC) logs saved during execution of the applications and represents the total bytes freed over all GC cycles.

We apply the static analysis to all these applications, and use our dynamic analysis to select the applications and candidate reuse sites to convert from the safe reuse sites found. As shown in Table 3, SPECjbb2005, xylem and lucene indicate the greatest potential for savings from object reuse for collections (including Strings and arrays). Hence we apply our automatic transformation to these applications and report the results in Table 4.

The following section presents our experimental results and analysis.

Table 1. Benchmarks analysed

Application	Description	Freed Memory (Object Churn)
SPECjbb2005	Server-side Java Benchmark	8 KB/txn
xylem	Proprietary tool to detect null references	1203MB
DaCapo lusearch	A text search tool	4913 MB
DaCapo pmd	A source code analyzer for Java	1178 MB
DaCapo ps	A postscript interpreter	2366 MB
DaCapo antlr	A parser generator and translator generator	884 MB

Table 2. Reuse site detection statistics

	SPECjbb	xylem	lusearch	pmd	ps	antlr
functions analysed	864	1679	2614	5587	1022	2486
statements analysed	864	33924	69688	126312	19078	100359
total analysis time	23s	33s	55s	8m 15s	20s	3m 16s
prelim analysis time	18s	25s	41s	3m 41s	16s	2m 29s
Results	SPECjbb	xylem	lucene	pmd	ps	antlr
no. of alloc sites	1014	1853	1549	2252	1013	2712
no. of alloc sites in loops	784	1456	776	1076	146	1852
no. of (safe) reuse sites found	251	400	688	577	77	375
no. of collection reuse sites	84	220	266	125	12	97
no. of string reuse sites	27	0	9	4	2	51
no. of sites reusable only along some paths	273	148	657	507	67	401
pure level 1 reuse sites	90	274	197	166	28	79
pure level 2 reuse sites	4	3	2	0	0	0
min level 1 reuse	280	416	766	640	93	477
min level 2 reuse	15	6	9	1	0	2

5.1 Reuse Site Detection Statistics (static analysis)

Table 2 summarizes the results of from the static analysis phase to find safe reuse sites and the closest enclosing loop where they may be reused. We notice that most opportunities exist at level 1 or level 2 reuse, and that a significant number of sites are only reusable along some paths and not others. Less than half of the safe reuse sites found are reusable at a single level along all paths. Except for ps, most benchmarks have a significant number of collection or string reuse sites.

Discussion: Analysis Time and Scalability. Table 2 also reports the times for analysis, and how much of that is spent on the preliminary analysis. We rely on an underlying context-sensitive flow analysis. This is, in general, slow, however, with suitable engineering, it can be reasonably scalable. Our analysis is built on top of a basic slicer which we have previously run on programs that are larger than 450,000 lines of code and the preliminary analysis took less than 10 minutes as reported in 5.

The additional analysis that we apply does an all-path exploration to determine the reusability of an object. This is clearly an exponential algorithm. pm�, at a little over 126K bytecode instructions analyzed, took 8 minutes and 15 seconds to analyze, of which the preliminary analysis took 3 minutes and 41 seconds. Here again, we use standard engineering tactics to contain the exponential state space exploration. If, for a given object, the analysis takes too long (currently curtailed at 30 seconds), we abort analysis of the object and conservatively mark it as not reusable.

Table 3. Reuse site object churn statistics

	SPECjbb	xylem	lusearch	pmd	ps	antlr
%churn at (safe) reuse sites	54%	18.4%	77.5%	16.4%	6%	14.6%
%churn at collection (and string) reuse sites	46.6%	16.5%	63%	5%	3.7%	4.25%
%churn at sites reusable only along some paths	6.8%	3.28%	77.2%	15.9%	6%	14%
no of reuse sites with more than 1% churn	7	3	22	9	5	12
no of collection reuse sites with more than 1% churn	5	2	8	1	2	1
%churn at top 3 reuse sites	48%	16%	46%	10%	5.8%	5.8%
Distribution of reuse levels	SPECjbb	xylem	lucene	pmd	ps	antlr
%churn at level 1 reuse sites	81%	99.98	36%	87.2%	50%	84%
%churn at level 2 reuse sites	18%	0.02	64%	12.4%	50%	16%
Distribution of reuse levels for collections	SPECjbb	xylem	lucene	pmd	ps	antlr
%churn at level 1 reuse sites	89%	99.98%	25%	42.8%	100%	99.3%
%churn at level 2 reuse sites	11%	0.02%	75%	57.2%	0%	0.7%

5.2 Reuse Site Object Churn Statistics (dynamic analysis)

Table 3 captures some of the statistics gathered during the dynamic analysis phase based on simple allocation profiling to identify reuse sites that generate more temporary objects.

We observe that in many cases, a few potentially reusable sites cause a perceptible amount of object churn, particularly in SPECjbb2005, lucene and xylem. The results also reflect the importance of being able to handle reuse sites which are safely reusable along some paths but not others. In some benchmarks, e.g. lusearch, the sites that are the top contributors to temporary objects bloat are of this nature.

5.3 Performance Impact Statistics

Object reuse conversion was applied only to the reuse sites that are indicated by dynamic analysis to have a major contribution to object churn. The performance comparisons between the original and converted application are presented in Table 4.

In general, the performance impact of reducing object allocations depends on the workload, choice of JVM used and both JVM and system parameters. For example, the JVM heap size, the garbage collection algorithm, system memory bandwidth characteristics (esp. on multi-core systems [6]) and workload specific tuning can affect results of comparisons. However, in our evaluation we focus on

Table 4. Performance impact statistics: The percentages are baselined against the corresponding results for the original benchmark without object reuse conversion; for example 100% reduction in temporary objects generated would mean that all object allocations were eliminated, 100% improvement in throughput would mean that throughput doubled.

SMALL INPUT SIZE	SPECjbb	xylem	lucene
No. of objects reused	24/txn	144851	232881
No. of element allocations reused	1920/txn	-	125750
% Reduction in temporary objects generated	41	22	24
% improvement in execution time or throughput	7.9	16.4	6.6
LARGE INPUT SIZE	SPECjbb	xylem	lucene
No. of objects reused	21/txn	342651	448787
No. of element allocations reused	1723/txn	-	250739
% Reduction in temporary objects generated	41	27	24
% improvement in execution time or throughput	21.6	19.9	6.2

the effectiveness of our technique rather than characterization of the degree of performance improvement expected from reducing object churn under different conditions. Hence we directly use default configurations instead of explicitly varying/tuning JVM and system parameters.

System Configuration. Our performance measurements were taken on a dual core Intel(R) Core(TM)2 Duo T7500, 2.2 GHz with 2GB RAM running Linux, Java HotSpot(TM) Server VM (build 14.3-b01, mixed mode). For the SPECjbb2005 measurements, we used an 8-core Intel server (Intel(R) Xeon(R) X5460, 3.16 GHz) with 16GB RAM, running Linux, Java HotSpot(TM) Server VM (build 1.6.0-b105, mixed mode).

JVM settings. For Xylem, we used a heap size of 1.6GB. We used out-of-the-box configuration parameters for the other benchmarks. In the case of SPECjbb2005, the heap size specified in the default benchmark properties file was 256MB. For the DaCapo benchmarks, the default heap size was as determined by the JVM. In all cases, the default garbage collection policy was determined by the specified JVM.

Since the execution time impact of reducing object creation can be highly dependent on the JVM and system parameters, we also measure other metrics like the percentage reduction in bytes of temporary objects used estimated from garbage collection statistics and relative scaling with larger input sizes. This enables us to evaluate whether our transformation is efficient enough to exploit potential for performance gains where opportunities exist.

We observe 20-40% reduction in object churn with our transformation. The execution time improvements range between 6-20%.

In SPECjbb2005, a single heavy allocation site dominates the reuse counts. Despite the fact that this is a string object and there are overheads due to reflection and accessing thread local context, we see significant benefits from

object reuse automation. These improvements appear to be consistent with those reported for a manual implementation of object reuse by researchers of [6]; their results were for a well-tuned setup (large heap, GC tuning)⁷.

We note that execution time improvements do not uniformly reflect the percentage reduction in objects. As observed by previous researchers [7,6,8] the relationship between percentage reduction in objects and performance is complex and depends on many factors ranging from workload and program characteristics, object construction costs, JVM tuning and hardware/system characteristics.

6 Related Work

Object churn analysis, impact and solutions. Many compiler and runtime optimizations like escape analysis [9,10,11,12], escape detection and improvements in memory management and garbage collection techniques [13] have been developed to reduce the overheads of allocating and reclaiming temporary objects. As part of their work on escape analysis, Blanchet [10] consider the problem of stack size limitations in using stack allocation for loop objects and implement a simple liveness check to enable reuse of stack allocated space in loops. Their solution however does not consider higher levels of reuse in nested loops. They also rely on the use of inlining in case the loop header and allocation site are not within the same method, which is not practical in framework based applications where the allocation may lie several levels deep in the call chain from the closest enclosing loop.

Shankar et al [7] found that even a sophisticated escape analysis implementation in a high performance production JVM typically eliminates less than 10% of allocations in component based applications. They experimented with the use of aggressive guided inlining of regions with high object churn to enable the JIT to detect more opportunities for stack allocation of objects. In contrast to their approach we use static analysis approach to perform source code transformations for object reuse, which enables us to detect additional opportunities without incurring a runtime overhead.

Performance understanding techniques have been proposed [14,15] for guiding programmers in eliminating excess temporaries that cannot be automatically detected by runtime optimizers. For example Buytaert et al [15] identify locations where code refactoring can be applied to reduce object creations. While their goal is similar to ours, they do not propose automated transformations. Their detection scheme uses dynamic traces unlike our static analysis approach where the dynamic analysis phase is only used to estimate potential benefits from the conversion.

Other approaches that help reduce the impact of excess temporary objects include advancements in memory management techniques for ensuring faster

⁷ [6] Also reports results of experiments conducted across a whole range of JVM settings (heap size, GC policies) to show that performance degradation from excessive object allocation in this case is not a mere artifact of GC algorithm or JVM parameters.

reclamation or reuse of temporary objects, e.g taking better advantage of allocation phases in the application [16], or combining the benefits of explicit object release [17,18,8] with garbage collection or scoped batch reclamation. Our technique is complementary to these efforts as it avoids the creation of objects wherever possible.

Zhao et al [6] analysed the implications of object allocation on scalability and performance. They proposed the notion of an allocation wall that limits multi-core scalability programs that perform high volumes of temporary objects. For their experimentation they perform manual code modifications to implement a form of object pooling for objects that are allocated very frequently and showed significant benefits for SPECjbb and SPECjvm derby. In their paper they observe that the process of manually converting an application for object reuse is time consuming and hence impractical for application developers to use. Our work succeeds in efficiently automating such optimizations for collection objects and strings.

Analysis and measurement of software bloat. Mitchell, Sevitsky and Srinivasan [19] define metrics based on modeling runtime information flow to classify and characterize the nature and volume of data transformations executed, though these measures have not been automated till date. The notion of data structure health signatures proposed by Mitchell and Sevitsky [20] has been used very effectively in characterization and automated measurement [21] of Java memory bloat in long lived heap objects. This is a relative measure of total memory bytes consumed by actual data vs associated representational memory overhead. For some categories of bloat, including the problem of temporary objects bloat which we address in this paper, an explicit model may not always be available for distinguishing overhead from necessary data or activity. Researchers have therefore used different measures of excesses like excessive volumes of temporary objects, data copies and heavy object creation costs to recognize the presence of bloat. For example, Xu et al use an instrumented JVM to summarize chains of runtime data copies [22] and an abstract thin dynamic slicing technique to identify data structures with high cost-benefit ratios [23]. Most approaches for detecting bloat have employed dynamic analysis. [24] applies a static analysis scheme to detect inefficient uses of container objects, particularly for underpopulated and overpopulated containers. All of these techniques are focused on aiding the process of reducing bloat, however, they are intended for interpretation by experts, not as fully automated solutions to de-bloat software like ours.

Dufour et al [14] apply blended static and dynamic analysis techniques to runtime traces for characterizing the usage of temporaries. Their results show that a significant number of temporary objects may be used several call levels away from their allocation site, which makes them particularly difficult to optimize. This motivates the need for techniques like ours.

7 Conclusion and Future Work

We presented an analysis technique to automatically detect and convert opportunities for object reuse in Java programs where there is significant potential

for benefit from reuse. This is a challenging problem because an object may be reusable in loops that may be several levels above it in the callgraph. Further, as our empirical results show, very often objects may be reusable only along certain paths and not others. In this situation a conservative analysis can miss most opportunities for reuse. We are able to improve precision in such situations by checking whether the conditions required for the correctness of our runtime transformation are met in the event these paths share the same loop header. Our results show that this solution can detect such opportunities in real large programs and reduce the generation of temporary objects significantly.

Further improvements in scalability and precision of our solution can be attained by incorporating feedback from our dynamic analysis to focus static analysis on the allocations sites that are likely to yield most benefits. Other future work includes extending the applicability of our automated transformation to other types of objects and using a combination of byte code manipulation and JVM level optimizations to improve the performance of the transformed code.

Acknowledgments. We thank Gary Sevitsky, Matt Arnold, Kazuaki Ishigaki, Dibyendu Das, Vijay Mann and Prasanna Kalle for their help, particularly their contribution to discussions on the problem of Java temporary objects bloat that motivated this work. We also thank Rupesh Nasre, and our anonymous reviewers for their excellent feedback on the paper.

References

1. Mitchell, N., Schonberg, E., Sevitsky, G.: Four trends leading to java runtime bloat. *IEEE Software* 27(1), 56–63 (2010)
2. Xu, G., Mitchell, N., Arnold, M., Rountev, A., Sevitsky, G.: Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In: *Future of Software Engineering Research* (2010)
3. Shirazi, J.: *Java performance tuning*. O'Reilly, Sebastopol (2003)
4. Blackburn, S., et al.: The DaCapo benchmarks: Java benchmarking development and analysis. In: *Object-Oriented Programming, Systems, Languages and Applications, OOPSLA* (2006)
5. Nanda, M.G., Sinha, S.: Accurate interprocedural null-dereference analysis for Java. In: *Proc. of the 31st Intl. Conf. on Softw.*, pp. 133–143 (2009)
6. Zhao, Y., Shi, J., Zheng, K., Wang, H., Lin, H., Shao, L.: Allocation wall: a limiting factor of java applications on emerging multi-core platforms. In: *Object-Oriented Programming, Systems, Languages and Applications, OOPSLA* (2009)
7. Shankar, A., Arnold, M., Bodik, R.: Jolt: lightweight dynamic analysis and removal of object churn. In: *Object-Oriented Programming, Systems, Languages and Applications, OOPSLA* (2008)
8. Guyer, S., McKinley, K., Frampton, D.: Free-me: A static analysis for automatic individual object reclamation. In: *Programming Language Design and Implementation, PLDI* (2006)
9. Choi, J.-D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for java. *SIGPLAN Not.* 34(10), 1–19 (1999)
10. Blanchet, B.: Escape analysis for object-oriented languages: application to java. *SIGPLAN Not.* 34(10), 20–34 (1999)

11. Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: International Conference on Compiler Construction (2000)
12. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for java programs. SIGPLAN Not. 34(10), 187–206 (1999)
13. Bacon, D.F., Cheng, P., Rajan, V.T.: A unified theory of garbage collection. SIGPLAN Not. 39(10), 50–68 (2004)
14. Dufour, B., Ryder, B.G., Sevitsky, G.: A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In: SIGSOFT 2008/FSE-16, pp. 59–70 (2008)
15. Buytaert, D., Beyls, K., De Bosschere, K.: Hinting refactorings to reduce object creation in java. In: ACES, pp. 73–76 (2005)
16. Xian, F., Srisa-an, W., Jiang, H.: Microphase: an approach to proactively invoking garbage collection for improved performance. In: Object-Oriented Programming, Systems, Languages and Applications, OOPSLA (2007)
17. Cherem, S., Rugina, R.: Uniqueness inference for compile-time object deallocation. In: ISMM (2007)
18. Inoue, H., Komatsu, H., Nakatani, T.: A study of memory management for web-based applications on multicore processors. In: Programming Language Design and Implementation, PLDI (2009)
19. Mitchell, N., Srinivasan, H.: Modeling runtime behavior in framework-based applications. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 429–451. Springer, Heidelberg (2006)
20. Mitchell, N., Sevitsky, G.: The causes of bloat, the limits of health. In: Object-Oriented Programming, Systems, Languages and Applications, OOPSLA (2007)
21. Mitchell, N., Schonberg, E., Sevitsky, G.: Making sense of large heaps. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 77–97. Springer, Heidelberg (2009)
22. Xu, G., Arnold, M., Mitchell, N., Rountev, A., Sevitsky, G.: Go with the flow: profiling copies to find runtime bloat. In: Programming Language Design and Implementation, PLDI (2010)
23. Xu, G., et al.: Finding low-utility data structures. In: Programming Language Design and Implementation, PLDI (2010)
24. Xu, G., Rountev, A.: Detecting inefficiently-used containers to avoid bloat. In: Programming Language Design and Implementation, PLDI (2010)

A Featherweight Approach to FOOL

Atsushi Igarashi

Department of Intelligence Science and Technology,
Graduate School of Informatics, Kyoto University, Japan

Abstract. It has always been challenging research topics to lay solid foundations such as formal semantics and type systems for object-oriented programming languages. It is challenging because advanced language constructs for high-level abstraction have inherent (and often unintended) complexity and the interaction caused by their combinations can be very wild. One effective approach to the complexity and wildness is to focus only on a relevant language core, which makes the problem tractable. This talk will discuss benefits and also limitations of such a “featherweight” approach to foundations of object-oriented languages (FOOL), along with a review of my work in this research area, and draw some lessons I’ve learned to make it work.

Related Types

Johnni Winther and Michael I. Schwartzbach

Department of Computer Science, Aarhus University
{jw,mis}@cs.au.dk

Abstract. We have identified a large class of Java errors that are caused by method calls becoming trivial since their formal and actual argument types are unrelated. A classical example is a call to the `equals` method on a `String` object with an `Integer` argument, which always returns `false`, but similar situations occur for many other methods in common frameworks. Typically, the programmer has provided the wrong arguments and meant for the call to have a non-trivial effect. We give a thorough analysis of the bug patterns found through our experiments which show that such errors are not only frequent in development code but also persist in production code. We formalize the notion of relatedness between actual and formal types and capture this in an extension of the Java 1.6 type system. The resulting compiler, which is fully implemented and backwards compatible, performs a static analysis that is modular and efficient.

Keywords: Java 1.6, dead code, bug finding, type checking.

1 Introduction

Certain method calls in Java are legal according to the type rules but are most likely errors, since the types of the actual and formal arguments are actually unrelated. A classical example is a call to the `equals` method on a `String` object with an `Integer` argument. The call is legal, since the formal argument type of the method is `Object`, but the result will always be `false`. It is unlikely that the programmer intended to perform this trivial call, and from a software engineering perspective it should be flagged by the compiler.

We have identified a large class of similar errors that are caused by unrelated types inducing unintended trivial calls. These situations are invisible to the compiler and often do not cause runtime errors either. Instead, the calls silently fail to perform a desired effect and thus degrade either the efficiency or the precision of the computation. A completely disastrous example involves dynamic programming, where the table lookup always fails because the `get` method in a `Map` is called with the wrong type of argument and thus always returns `null`. This will bypass the whole optimization and instead of the intended polynomial speed-up leave the implementation with an exponential complexity.

Our experiments show that such errors are common in code during development and even persist in production code like OpenJDK, Apache and Eclipse. We have performed a thorough analysis of the bug patterns which often

involve long and confusing dot sequences, like `a.b.c.d`, where the programmer gets lost in the class hierarchy, or makes failed attempts to use the type checker for refactoring.

We extend the Java type system with a notion of *related types* expressed through annotations which allow the programmer of a library to annotate methods with requirements for relatedness of formal and actual argument. For instance give the `equals` method on `String` the signature `equals(~String)` to declare that the method has only a trivial effect if called with an argument that is not a `String` instance. Invocations of such methods are then checked to ensure that the argument could be of the intended type. Note that client code need *not* change for this to work. The annotations added to the library code will alone enforce the added checks.

Our type system is not conventional. Normally type systems are conservative but ours is *permissive*. A conservative type system would ensure that there are no such trivial call sites by disallowing all call sites where the formal and actual types are *potentially* unrelated. Such an approach would break existing code in that it would render otherwise valid code invalid. In contrast, our type system ensures backward compatibility by only disallowing call sites when we are *sure* that the formal and actual types are unrelated. We therefore do *not* provide any guarantees that all trivial calls are disallowed but as our experiments show all the disallowed calls do cause errors.

Since the annotations are built into the type system the judgements are based on an open-world assumption. This is in contrast to the tool FindBugs, which detects some of the bugs we are interested in, but which works in a closed-world assumption and where the frameworks that are considered are hardwired by tool developers.

The rest of this article is structured as follows: In Section 2 we define the targeted class of errors and in Section 3 we discuss related work. In Section 4 we formally define relatedness, in Section 5 we introduce the related type and in Section 6 we give the formal rules for integrating related types into Java. In Section 7 we present our experiments with related types. In Appendix A we present in detail some of the errors from our experiments.

2 Unintended Dead Code

A class of errors in Java can be categorized as *unintended dead code bugs*. In this section we explore such common errors and give a precise definition of the error class.

The `equals` method on the topmost Java class `Object` has the signature `boolean equals(Object)` so the following is valid Java:

```
void m(String str, Integer num) {
    if (str.equals(num)) {
        // dead code
    }
}
```


The JavaDoc [16] for `equals` on `String` states that the “result is true . . . only if the argument is not null and is a `String` object . . .”. Since the argument `num` is either null or an `Integer` object, and thus not a `String` object, the invocation will always return `false` and the body of the if-statement will never be executed. It is *dead code*.

Not only the `equals` method is subject to unintended dead code bugs. Several methods in the Java collection framework exhibit similar problems. For instance calling the boolean `contains(Object)` method on a `Collection<E>` instance can give a similar situation where the call always returns `false`¹:

```
boolean m(List<String> list, Integer num) {
    return list.contains(num);
}
```

These errors cannot be caught within the current Java type system nor through the introduction of a self-reference type like in [3,23]. Consider these variables:

```
String aString; Number aNumber; Integer anInteger;
List<?> someList; ArrayList<Integer> intList; LinkedList<Integer> intLinkedList;
```

For these variables, the following expressions should type check as follows:

```
anInteger.equals(aNumber); // should be type correct
aString.equals(aNumber); // should fail
intList.equals(intLinkedList); // should be type correct
someList.contains(aString); // should be type correct
intList.contains(aNumber); // should be type correct
intList.contains(aString); // should fail
```

For the typing of `equals` and `contains` we have the following options

```
class Object<E> {
    boolean equals1(Object e);
    boolean equals2(Self e);
}
interface List<E> {
    boolean contains1(Object e);
    boolean contains2(E e);
}
```

where `equals2` is typed using the self-reference type, `Self`, which is not part of the current Java type system. None of the above gives the wanted behaviour since we would have the following results:

```
anInteger.equals2(aNumber); // type error: Number <: Integer
aString.equals1(aNumber); // type correct
intList.equals2(intLinkedList); // type error: LinkedList<Integer> <: ArrayList<Integer>
someList.contains2(aString); // type error: String <: capture-of-?
intList.contains2(aNumber); // type error: Number <: Integer
intList.contains1(aString); // type correct
```

Even though this class of errors is related to the static type of the arguments it is not the usual kind of typing errors. The program breaks at neither compile-time nor runtime. On the contrary, implementations of such methods as `equals`

¹ Except in the special case where `list` contains a null element and `num` is null; an intention which the static types of `list` and `num` do not express.

and `contains` should always handle the case when the given object is of an “unrelated” type. Such errors are hard to discover since we need to observe their undesired effects at runtime. Often we need to find the error cause through debugging only to observe that the *compiler* has call site information that enables *us* to detect the error: For instance in the invocation `str.equals(num)` the compiler knows the static type of `str` to be `String` and of `num` to be `Integer`, and *we* know that a `String` can never be equal to an `Integer`.

We observe that many such call sites are *trivial call sites* defined as follows.

Definition 1 (Trivial Call Site). *We define a trivial call site as a method invocation for which all runtime calls have no side effects and returns the same fixed value, if any, as a consequence of the relation between the static types of the arguments and the receiver. In this definition we regard throwing a fixed exception, checked or not, as returning a fixed value.*

A trivial call site is thus the cause of an *unintended dead code bug* in that it renders the method invocation itself or the handling of its return value obsolete.

Much like the rule for reference equality in Java [12, §15.21.3] our new typing does not provide additional type safety properties but instead disallows usage that is most likely erroneous. For instance in this method

```
boolean m(String s, Integer i) {
    return s == i; // illegal
}
```

the expression `s == i` is illegal and this is neither because the expression would result in a runtime error nor that the result will always be `false`² but because, given the static types of the operands, the equality test is most likely unintended.

In the same way our type system performs an approximation of trivial call sites in order to disallow calls that most likely cause unintended dead code bugs. On the assumption that all methods annotated with related types are in fact trivial on unrelated argument types we thus state the following conjecture for our extended type system:

Conjecture 1. All call sites that are disallowed through the use of related types are trivial in all possible environments and are therefore most likely the cause of an unintended dead code bug.

3 Related Work

In the FindBugs [22] analysis tool two so-called detectors look for similar unintended dead code bugs. The EC detector checks the correctness of `equals` calls which includes checking for “relatedness” between the receiver and argument types. The notion of relatedness is not formalized but resembles relatedness as defined in this article in that it looks for relations not only at the type level, for instance between `Integer` and `String`, but also at the type argument level, for

² Both values could be null.

instance between `List<Integer>` and `List<String>`. A key difference is that FindBugs uses a closed world assumption thus giving only warnings when no two types proving the relation were found. Our related types operate under an open world assumption, giving errors when types are unrelated and providing warnings when types are only semi-related.

The GC detector in FindBugs checks for the use of “appropriate” types in calls to the generic collection classes. An “appropriate” type is not formalized but seems close to the relatedness of the EC detector. The lack of a formalized notion of relatedness seems to be the reason why we found two warnings with related types not found by FindBugs (See Section 7 and A.3). In these cases, calls to collection classes were made with unrelated arguments that would have been detected if the EC rules for relatedness were used.

The main difference though between FindBugs and our approach is that FindBugs is hardwired to specific methods in the standard Java API whereas related types can be used whenever the programmer sees fit. There are for instance several other collection frameworks [9,27,217] that could benefit from an improved handling of argument relatedness. With related types the programmers are given the tools to improve their own code instead of having to rely on an external analysis tool to be updated to their benefit.

Several tools exist for detecting dead code [25,8,13,10,11,19]. None of these base their detection on the semantics of individual call sites and they are therefore unable to detect unintended dead code bugs.

In the Java language extension MultiJava [6] the binary method problem [214] involving `equals` is addressed through multiple dispatch. MultiJava changes the semantics of dispatch to invoke the most precise method given the runtime argument types whereas we disallow the existing invocations that are trivial.

4 Relatedness

4.1 Related vs. Unrelated

Unintended dead code bugs arise because we invoke methods with ‘unrelated’ types making the call sites trivial. But what do we mean by ‘unrelated’? A clear example is that `String` and `Integer` are ‘unrelated’ since they have only the runtime value `null` in common but (un)relatedness can be more complicated, as shown by this generic pair type:

```
class Pair(X,Y) { X first; Y second; }
```

What values can an expression of static type `Pair<String,Number>` evaluate to at runtime? If the expression is not `null` we know that it must be an instance of a subclass of `Pair`. Furthermore we know that the fields `first` and `second` must be either `null` or a `String` object and an instance of a subclass of `Number`, respectively. We see that subtyping not only at the top level but deeply through the type arguments characterize the possible runtime values and that `null` occurs as a possible value at all levels.

Comparing `Pair<String,Number>` to `Pair<Object,Integer>` we see a possible common value in that `first` could be a `String` instance and `second` an `Integer` instance and for this reason we will say that the two types are related. On the other hand comparing `Pair<String,Number>` to `Pair<Integer,String>` we see that only if `first` and `second` are null can the two types hold the same instance. When this is the case we say that the two types have only *trivial values* in common.

That null is a possible common value does not provide any useful information about the relation between two types. We are interested in whether or not two types intend to hold the same runtime values. This is not something new to Java. Consider these methods:

```
boolean m1(String s, Integer i) { return s == i; /* invalid */ }
boolean m2(List<String> ls, List<Integer> li) { return ls.equals(li); }
```

The test for equality in `m1` is rejected in Java because `String` is not a subtype of `Integer` or vice versa; even though both expressions could be null. In the same spirit we regard the lists in `m2` as being of unrelated types since they can only exhibit equality through trivial values, i.e. by containing the same number of mere null elements. This choice is also confirmed by our experiments, described in Section 7, in which we have not encountered any need for such a trivial relatedness.

In the following sections we formalize the notion of relatedness through two relations on reference types, *relatedness* and *value subtyping*. The first relation defines when to reference types are related and the second is a stricter relation that defines the relation between the *related types* described in Section 5.

For the formalization and throughout this paper we use the formal definition for Java types given by the grammar³ in Figure 1. We let $T, S,$ and U range of reference types, $C, D,$ and E range over class/interface names, and A and B range over argument types.

In the formalization of these relations we need three auxiliary functions:

1. The $nm(T)$ function collects the class/interface names of the types that T extends. For instance $nm(\text{Collection}\langle\text{String}\rangle) = \{\text{Collection}, \text{Iterable}, \text{Object}\}$.
2. The $bd(C\langle\bar{A}\rangle)$ function takes a class/interface type $C\langle\bar{A}\rangle$ and returns the same class/interface type in which the type arguments have been replaced by their *reference bounds* defined as follows: For `? super T` the reference bound is T . For `? extends T` the reference bound is $T\&U$ where U is the bound of the type variable that `? extends T` replaces. For a related type $\sim T$, presented in Section 5, the reference bound is T , and for all other types T , $bd(T)$ is the identity function. This means that we for instance have $bd(\text{Map}\langle?\text{ extends Number},?\text{ super String}\rangle) = \text{Map}\langle\text{Number},\text{String}\rangle$.
3. The $sa(D, C\langle\bar{A}\rangle)$ function computes the list of type arguments of the D super type of $C\langle\bar{A}\rangle$. For instance $sa(\text{Comparable}, \text{Integer}) = [\text{Integer}]$ since `Integer` implements `Comparable<Integer>`.

³ We use the barred notation, \bar{T} , for a list or set of elements T_1, \dots, T_n for some $n \geq 0$.

Types	$\tau ::= \mathbf{N}$ \perp	normal type null type
Normal Types	$\mathbf{N} ::= \pi$ \mathbf{T}	primitive type reference type
Reference Types	$\mathbf{T} ::= \mathbf{C}(\overline{\mathbf{A}})$ $\mathbf{N}[]$ \mathbf{V} $\mathbf{T} \& \mathbf{T}$	class/interface type array type type variable intersection type
Argument Types	$\mathbf{A} ::= \mathbf{T}$ $? \text{ extends } \mathbf{T}$ $? \text{ super } \mathbf{T}$	reference type wildcard extends wildcard super

Fig. 1. Grammar for Java Types

4.2 Relatedness

Two reference types, \mathbf{T} and \mathbf{S} , are deemed related if there exists a non-trivial runtime value that can be viewed as an instance of both \mathbf{T} and \mathbf{S} . The relatedness relation is of the form $\Delta \vdash \mathbf{T} \sim \mathbf{S}$ where Δ is a type environment containing the bounds of the type variables. The relation is defined through the inference rules shown in Figure 2. For brevity, the symmetric versions of the asymmetric rules have been omitted.

The rules R-ARRBASE, R-ARRPRIM, and R-ARRREF define the relatedness for array types. The latter defers the relatedness of non-primitive array types to their base types, and likewise the rule R-TYPEVAR defers the relatedness of type variables to their bounds.

The rule R-CLSINT defines the relatedness for class/interface and intersection types. The rule has 3 premises. First, the first two lines in the premise define that there must be a *most specific class declaration* between the two types. For instance the most specific class declaration between `Number` and `Integer` is `Integer` whereas `Number` and `AbstractCollection(E)` have no most specific class declaration as they are both classes but neither is a subclass of the other. All interfaces have a most specific class declaration, namely `Object`.

Secondly, the third line in the premise defines that if one of the two types is final then it must be a subdeclaration of all declarations. Note that this is not the same as requiring that the final type must be a subtype of the other type. For instance, the class `final class NumList implements List(Num)` is related to `List(Integer)` even though `NumList` is not a subtype of `List(Integer)`. This is because a `NumList` instance might contain only `Integer` elements and they therefore have a common runtime value.

Thirdly, the fourth line defines that for all common supertypes of the two class/interface types the bounded type arguments must be related. For instance `List(Num)` and `Set(Integer)` are subtypes of `Collection(Num)` and `Collection(Integer)`, respectively, and are related because `Num` and `Integer` are related. Similarly `Set(String)` and `Set(Integer)` are not related since `String` and `Integer` are not related.

$$\begin{array}{c}
 \frac{T \in \{\text{Object, Cloneable, Serializable}\}}{\Delta \vdash \tau[] \sim T} \quad (\text{R-ARRBASE}) \\
 \\
 \Delta \vdash \pi[] \sim \pi[] \quad (\text{R-ARRPRIM}) \quad \frac{\Delta \vdash T \sim S}{\Delta \vdash T[] \sim S[]} \quad (\text{R-ARRREF}) \\
 \\
 \frac{\Delta(V) = U \quad \Delta \vdash U \sim S}{\Delta \vdash V \sim S} \quad (\text{R-TYPEVAR}) \\
 \\
 \frac{\begin{array}{l} C_1 = \text{classOf}(T) \quad C_2 = \text{classOf}(S) \\ i \in \{1, 2\}. \text{nm}(C_i) \supseteq \text{nm}(C_1) \cup \text{nm}(C_2) \\ C \text{ final} \in \text{nm}(T) \cup \text{nm}(S) \Rightarrow \forall C' \in \overline{D}. C' \in \text{nm}(C) \\ \forall C \in \text{nm}(T) \cap \text{nm}(S). \overline{T'} = \text{sa}(C, \text{bd}(T)), \overline{S'} = \text{sa}(C, \text{bd}(S)). \forall i. \Delta \vdash T'_i \sim S'_i \end{array}}{\Delta \vdash T \sim S} \quad (\text{R-CLSINT}) \\
 \\
 \text{classOf}(T) = \begin{cases} C & \text{if } T = \text{class } C(\overline{A}) \\ \text{Object} & \text{if } T = \text{interface } C(\overline{A}) \\ C_i & \text{if } T = S \& U, C_1 = \text{classOf}(S), C_2 = \text{classOf}(U), \\ & \text{and } \text{nm}(C_i) \supseteq \text{nm}(C_1) \cup \text{nm}(C_2) \end{cases}
 \end{array}$$

Fig. 2. Rules for relatedness, $\Delta \vdash T \sim S$

Determining relatedness between two types is inherently cyclic. Since the last rule, R-CLSINT, recursively requires the relation on the bounded type arguments, the result is self-dependent on self-bounded type variables. For instance, checking $\Delta \vdash \text{Enum}(X) \sim \text{Enum}(Y)$ with $\Delta(X) = \text{Enum}(X)$ and $\Delta(Y) = \text{Enum}(Y)$, the R-CLSINT rule requires $\Delta \vdash X \sim Y$ which in turn, through R-TYPEVAR, requires $\Delta \vdash \text{Enum}(X) \sim \text{Enum}(Y)$. This cyclic dependency is easily resolved by adding cycle detection to the algorithm and coinductively assuming the relation when a cycle is detected.

The \sim relation is symmetric but *not* transitive. The top type Object is related to all types, so we have Integer \sim Object and Object \sim String but Integer $\not\sim$ String since Integer and String have no common most specific class declaration.

4.3 Value Subtype

A type T is a *value subtype* of S if there exist expressions of static type S which evaluate to values of type T . This relation is a strict superset of ordinary subtyping on non-null reference types.

The value subtype relation is of the form $\Delta \vdash T \subset: S$ where Δ is a type environment containing the bounds of the type variables. The inference rules are shown in Figure 3.

The VS-ARRBASE, VS-ARRPRIM, and VS-ARRREF rules define value subtyping like ordinary subtyping, with VS-ARRREF deferring value subtyping covariantly to the array base types.

$$\begin{array}{c}
\frac{S \in \{\text{Object}, \text{Cloneable}, \text{Serializable}\}}{\Delta \vdash \tau[] C : S} \quad (\text{VS-ARRBASE}) \\
\\
\Delta \vdash \pi[] C : \pi[] \quad (\text{VS-ARRPRIM}) \quad \frac{\Delta \vdash T C : S}{\Delta \vdash T[] C : S[]} \quad (\text{VS-ARRREF}) \\
\\
\frac{\Delta(V) = U \quad \Delta \vdash T C : U}{\Delta \vdash T C : V} \quad (\text{VS-TYPEVARR}) \\
\\
\frac{\Delta(V) = U \quad \Delta \vdash U C : S}{\Delta \vdash V C : S} \quad (\text{VS-TYPEVARL}) \\
\\
\frac{\forall E \in nm(S). \overline{T'} = sa(E, bd(T)), \overline{S'} = sa(E, bd(S)). \forall i. \Delta \vdash T'_i C : S'_i}{\Delta \vdash T C : S} \quad (\text{VS-CLSINT})
\end{array}$$

Fig. 3. Rules for value subtyping, $\Delta \vdash T C : S$

For type variables the rules VS-TYPEVARL and VS-TYPEVARR define the value subtype relation to be deferred to the type variable bounds. Consider a type variable X with bound **Number**. X can hold the same values as **Number** and X is therefore a value subtype of any type of which **Number** is a value subtype. Likewise any type which is a value subtype of **Number** is a value subtype of X .

The last rule, VS-CLSINT, defines value subtyping for two class/interface or intersection types, T and S . Here we have two requirements. The first line in the premise defines that the type declarations of T must extend the type declarations of S . For instance $nm(\text{List}\langle\text{Integer}\rangle) \supseteq nm(\text{Collection}\langle?\text{ super Number}\rangle)$ because **List** extends **Collection**. The second line in the premise defines that for all common super type declarations of T and S , all type argument bounds of the corresponding supertypes must be pair-wise related under the value subtype relation. For instance **Collection** is a common super type declaration of **List** $\langle\text{Integer}\rangle$ and **Collection** $\langle?\text{ super Number}\rangle$ with super types **Collection** $\langle\text{Integer}\rangle$ and **Collection** $\langle?\text{ super Number}\rangle$, respectively. The bounds of the type arguments, here **Integer** and **Number** respectively, are required to be related by value subtyping, which they indeed are. This yields **List** $\langle\text{Integer}\rangle \subset: \text{Collection}\langle?\text{ super Number}\rangle$ which expresses that **Collection** $\langle?\text{ super Number}\rangle$ can hold a list of elements all being **Integer** instances.

Just like relatedness, determining value subtyping between two reference types is inherently cyclic. Again the cyclicity is resolved coinductively by assuming value subtyping on self-dependency. Unlike the subtype relation in Java, we know the value subtyping relation to be decidable. The left and right terms are never swapped which implies decidability, cf. [17, Section 5.1].

As one might expect, the value subtype relation is transitive as stated by the following theorem:

Theorem 1 (Value Subtype Transitivity). *For all types T, S, U and environments Δ , if $\Delta \vdash T \subset: S$ and $\Delta \vdash S \subset: U$ then $\Delta \vdash T \subset: U$.*

Proof. By induction in the structure of T and S .

4.4 Relatedness and Value Subtyping

The value subtyping relation is a subset of relatedness and thus value subtyping implies relatedness. Although relatedness is not transitive in itself, we have the weaker property of transitivity through the value subtype relation as stated in the following theorem.

Theorem 2 (Weak Transitivity). *For all types T, S, U and environments Δ . If $\Delta \vdash T \sim S$ and $\Delta \vdash S \subset: U$ then $\Delta \vdash T \sim U$.*

Proof. By induction in the structure of S .

Note that this transitivity is directional: If T is a value subtype of S and S and U are related, then T and U might not be related: $\text{FutureRunnable}\langle\text{String}\rangle \subset: \text{Runnable}$ and $\text{Runnable} \sim \text{FutureRunnable}\langle\text{Integer}\rangle$ but $\text{FutureRunnable}\langle\text{String}\rangle \not\sim \text{FutureRunnable}\langle\text{Integer}\rangle$ due to their unrelated type arguments String and Integer .

Theorem 2 defines the way relatedness can be composed and several type rules described in Section 6 are based on this property.

5 Type Annotations

5.1 The Related Type

To incorporate the notion of relatedness into Java we extend the set of reference types [12, §4.3], here called *original* reference types, with the *related type* $\sim T \triangleleft U$ for every original reference type T and U . The grammar for Java types in Figure 1 is therefore extended to include related types by replacing reference types with the grammar shown in Figure 4. For a related type $\sim T \triangleleft U$, we call T the *intended type* of $\sim T \triangleleft U$ and U the *bound* of $\sim T \triangleleft U$. The bound is described in Section 5.4 and if omitted we take the bound to be `Object`. For instance $\sim\text{String}$ is the type “related to `String`” with intended type `String` and implicit bound `Object`.

A related type can be used anywhere a type variable can be used, except in the `throws` clause of a constructor/method declaration. Declaring that a method

Reference Types	$T ::= T_0$ $\sim T_0 \triangleleft T_0$	original reference type related type
Original Reference Types	$T_0 ::= C(\overline{A})$ $N[]$ V $T \& T$	class/interface type array type type variable intersection type

Fig. 4. Grammar for Reference Types with Related Types

throws a type related to `T` would be the same as declaring that it throws `Throwable` as we have no other certain knowledge about what might be thrown.

The most interesting usage of a related type is as an argument type. By declaring a method with signature `m(~String)` we state that the invocation is *trivial* if called with a type not “related to `String`”. In other words the *intended* argument type is `String`. Note that it is up to the implementer not to declare an intended type that is too strict since no checks are made to ensure that the method implementation is actually trivial on unrelated argument types.

What *is* checked is that no calls are trivial, i.e. that the argument types have possible non-trivial values at runtime. For instance the method `m(~String)` will be applicable only if the argument type is related to the intended argument type `String`, that is if the argument *could* be a `String` instance.

5.2 Examples of Use

The related type annotation can be used for instance to improve the signature of `equals` on `String` to `boolean equals(~String)`. As mentioned earlier, the JavaDoc implies this contract on the argument type; that it is intended to be a `String` instance. Similarly `equals` on `Integer` can be refined to `boolean equals(~Integer)`, on `Long` to `boolean equals(~Long)` and so forth.

Also many interfaces redeclare `equals` to refine the intended argument type: `Collection<E>` implicitly refines its intended argument type to `Collection<E>`, `List<E>` states that `equals` must return “true ... only if the specified object is also a list...” and similarly for the `Set<E>` and `Map<K,V>` interfaces. All `equals` signatures can therefore be improved by refining the argument types to `~Collection<E>`, `~List<E>`, `~Set<E>` and `~Map<K,V>`, respectively.

In the Java collections framework we find several methods other than `equals` that are trivial on unrelated arguments. For instance `contains` and `containsAll` on `Collection<E>` could be refined to use `~E` and `Collection<? extends ~E>`, respectively. Similarly for `remove`, `removeAll` and `retainAll` on `Collection<E>`, `indexOf` and `lastIndexOf` on `List<E>`, and `containsKey`, `containsValue`, `get` and `remove` on `Map<K,V>`.

It is not only in the Java Standard Library that we can make use of related types. There are several collection frameworks like `Javolution` [9], `fastutil` [27], `Joda Primitives` [7], and the `Google Collections Library` [2] for which the use of related types would improve the precision of the interfaces.

For instance the `Google Collections Library` contains a `multimap` interface `com.google.common.collect.Multimap<K,V>` with methods

```
boolean containsEntry(Object key, Object value);
boolean containsKey(Object key);
boolean containsValue(Object value);
boolean remove(Object key, Object value);
Collection<V> removeAll(Object key);
...
```

for which the use of type `~K` for all key arguments and `~V` for all value arguments would be appropriate.

5.3 Related Types and Variance

The use of a related type $\sim T$ as a type argument should not be confused with the use of wildcards. For instance `List< \sim Number>` is a list of elements that are assumed to be of a type related to `Number`. We can add values of a static type related to `Number` to the list but we cannot add a value of an unrelated type such as for instance a `String` object. Note that we only assume but do not guarantee that elements are of a type related to `Number`. We could for instance hold a `String` instance in an expression of static type `Serializable` and therefore be allowed to add it to the list.

For this reason, unlike wildcards, we have no variance on related types. `List<Integer>` is *not* a subtype of `List< \sim Number>` as this will allow contamination of the list elements [15,26]. Using wildcards, we do however have that both `List<Integer>` and `List<Object>` are subtypes of `List<? extends \sim Number>`. As defined in Section 6 we have that since both `Integer` and `Object` are related to `Number` these are subtypes of \sim Number; it makes sense to assume that the elements of such lists could be `Number` instances.

5.4 Bounds on Related Types

In the previous examples the related types replace the argument type `Object` but related types can have other upper bounds. Consider this example:

```
class A<X extends Number> {
    X getX() { ... }
    long m(A< $\sim$ Long> n) { return n.getX().longValue(); }
}
```

Here, the type variable `X` is declared with the bound `Number`. This means that in the method `m` we should at least know that `getX()` returns a `Number` instance, if not null. This implies that the related type \sim Long must carry the upper bound `Number`, here implicitly given by the bound of `X`.

Since the related types must always implicitly carry bounds we can extend the expressiveness of the formalism by allowing for explicit bounds without additional complexity. Using the syntax $\sim T$ extends `S` we can declare a related type to have the intended type `T` and upper bound `S`.

A use case for related types with bounds is the definition of an additional equivalence relation on a type subset. For instance defining a method `equalsRaw` method on a type hierarchy could yield the following interfaces:

```
interface Type {
    boolean equalsRaw(Type other);
}
class ArrayType implements Type {
    boolean equalsRaw( $\sim$ ArrayType extends Type other) { ... }
}
class ClassType implements Type {
    boolean equalsRaw( $\sim$ ClassType extends Type other) { ... }
}
```

Here `Type` declares its `equalsRaw` method to take an argument of type `Type` but the subtypes `ArrayType` and `ClassType` refine their intended arguments to be related to their respective types but with a bound of type `Type`. This can be generalized to larger scenarios.

6 Type Rules

In this section we formally define the rules that enable us to integrate related types into the Java type system.

Wellformedness. The wellformedness of a related type $\sim T \triangleleft U$ is defined by the WF-RELATED in Figure 5. The only requirement is that the bound, U , is related to the intended type, T . This ensures that the type has a non-trivial value set. For instance, the type $\sim \text{String} \triangleleft \text{Integer}$ is badly formed since no non-null value can both subtype `Integer` and be related to `String`.

Type Erasure. The type erasure [14] [12, §4.6] of a type T is used for compatibility with pre-generic versions of Java and the encoding of `.class` files. We define the type erasure of a related type $\sim T \triangleleft U$ to be the type erasure of U . This means that for instance the refinement of `boolean equals(~String)` on `String` will have the same method descriptor [18, §4.3.3] as the original method `boolean equals(Object)`, namely⁴ `(Ljava/lang/Object;)Z`. In context of the example in Section 5.4 this means that the method descriptor for the `equalsRaw` methods on `Type`, `ArrayType` and `ClassType` are all the same, namely `(LType;)Z`. This definition thus ensures the requirement for binary compatibility as defined in [12, §13.2].

Intersection. The definition of intersection of two types, T and S , denoted $T \& S$, found in [12, §4.9] is extended to include related types by the rules in Figure 6 where $T \& S =_{\Delta} U$ defines the intersection of T and S under type environment Δ to be U . The intersection of types occur in Java during capture [12, §5.1.10] and during type inference [12, §15.12.2.7].

We have two rules that define the intersection on related types. The first rule, I-REFERENCE, computes the intersection of an original reference type, S , and a related type, $\sim T \triangleleft U$. The resulting type is a new related type, $\sim T \triangleleft (U \& S)$, with the same intended type as $\sim T \triangleleft U$ but with the new bound $U \& S$. For instance in the example in Section 5.4 the intersection of `Number` and $\sim \text{Long} \triangleleft \text{Object}$ is $\sim \text{Long} \triangleleft \text{Number}$.

The second rule, I-RELATED, defines the intersection between two related types. This is defined as a new related type whose intended type is the intersection of the intended types and whose bound is the intersection of the bounds.

Subtyping. Related types are integrated into the Java subtype relation through the extension shown in Figure 7. The subtyping judgement is of the form $\Delta \vdash T <: S$, concluding that under the type environment Δ , the type T is a subtype of S . Subtyping must preserve the existing preservation and progress properties of the type system [14, 5]. Because of the override rules described later in this section we cannot always rely on the relatedness information of a related type.

⁴ The notation for method descriptors as used within `.class` files, defined in [18].

$$\frac{\Delta \vdash T \sim U}{\Delta \vdash \sim T \triangleleft U \text{ ok}} \quad (\text{WF-RELATED})$$

Fig. 5. Rules for wellformedness

$$\frac{\Delta \vdash T \sim S}{(\sim T \triangleleft U) \& S =_{\Delta} \sim T \triangleleft (U \& S)} \quad (\text{I-REFERENCE}) \qquad \frac{(\sim T \triangleleft U_1) \& (\sim S \triangleleft U_2)}{=_{\Delta} \sim (T \& S) \triangleleft (U_1 \& U_2)} \quad (\text{I-RELATED})$$

Fig. 6. Rules for intersection

$$\frac{\Delta \vdash U <: S}{\Delta \vdash \sim T \triangleleft U <: S} \quad (\text{S-BASE})$$

$$\frac{\Delta \vdash T <: U \quad \Delta \vdash T \sim S \quad \Delta \vdash T \not<: S \wedge \Delta \vdash S \not<: T \Rightarrow \text{Warning}}{\Delta \vdash T <: \sim S \triangleleft U} \quad (\text{S-RELATED})$$

$$\frac{\Delta \vdash U_1 <: U_2 \quad \Delta \vdash T <: S}{\Delta \vdash \sim T \triangleleft U_1 <: \sim S \triangleleft U_2} \quad (\text{S-SUBTYPE})$$

Fig. 7. Subtyping rules for related types, $\Delta \vdash T <: S$

$$\Delta \vdash T \sqsubset: T \quad (\text{AR-REFL}) \quad \Delta \vdash \sim T \triangleleft U \sqsubset: U \quad (\text{AR-BASE})$$

$$\frac{\Delta \vdash T \sim S \quad S \neq T \Rightarrow \Delta \vdash S \not<: T}{\Delta \vdash \sim T \triangleleft U \sqsubset: \sim S \triangleleft U} \quad (\text{AR-RELATED})$$

Fig. 8. Argument type replacability rules, $\Delta \vdash T \sqsubset: S$

Therefore we have no other certain knowledge about the runtime value of an expression of static type $\sim T \triangleleft U$ than it must be a U instance, if not `null`. The only rule relating a related type to an original reference type is therefore S-BASE, subsuming $\sim T \triangleleft U$ under U .

The second rule, S-RELATED, incorporates the notion of relatedness. An original reference type, T , is a subtype of $\sim S \triangleleft U$ if T is a subtype of U and if T is related to S . If T is not a value subtype of S or vice versa, we say that T and S are *semi-related*. For instance this is the case for `Serializable` and `Cloneable`; none is a value subtype of the other but they may still have common value subtypes, like for instance `java.util.BitSet`. If T and S are only semi-related a warning is issued during type checking. As shown in Section 7, semi-related types sometimes, but *not always*, indicate underlying errors and we therefore choose to generate warnings instead of errors in these cases.

Theorem 2 gives us that relatedness extends through the value subtype relation, that is, if T and S are related and S is a value subtype of U , then T and U are related. The third rule, S-SUBTYPE, uses this property to provide a subtype relation between related types provided that the bound of $\sim T \triangleleft U_1$ is also a subtype of the bound of $\sim S \triangleleft U_2$.

Override Rules. We define a set of rules for *argument type replacability* which are shown in Figure 8. The rules are of the form $\Delta \vdash T \sqsubset: S$ where Δ is a type variable environment and T, S are types. The judgement $T \sqsubset: S$ states that argument type T can replace S when overriding.

The first rule, AR-REFL, define all original reference types to be invariant in overriding. Because of method overloading in Java [12, §8.4.9], it is not possible to use contravariant argument types when overriding. Such a method would have a different signature and would therefore no longer override the inherited method. The definition of override-equivalent signatures in Java [12, §8.4.2] leaves room for mixing raw and generic types but we omit this in AR-REFL for brevity.

The second rule, AR-BASE, allows a related type, $\sim T \triangleleft U$, to replace its bound, U . This is possible because the type erasure of $\sim T \triangleleft U$ is U , thus ensuring binary compatibility between the replacing and replaced methods.

The third rule, AR-RELATED, provides a sanity check for replacement between related types, namely that they must not contradict each other. If the two intended argument types are not related then no value type will subtype both, i.e. what is intended for the one argument type is unintended for the other, and vice versa. Such a replacement is therefore prohibited. Additionally, we require that T is *not* a strict value supertype of S . The weak transitivity of Theorem 2 enables us to obtain the aforementioned unrelatedness indirectly. For instance replacing $\sim\text{String}$ with $\sim\text{Object}$ and then again with $\sim\text{Integer}$ creates a contradiction between the first and the third intended argument type. This additional requirement is thus a sanity check that makes it more difficult, though not impossible, to make contradicting replacements. To ensure binary compatibility we furthermore require the two related types to have the same bound. Since this rule allows for a replacement of for instance $\sim\text{Serializable}$ to $\sim\text{Cloneable}$ for which relatedness of one does not imply relatedness of the other we can not rely on relatedness in the subtyping rule S-RELATED.

As mentioned in Section 5.2, many types redeclare the `equals` method indicating that the intended argument type is a refinement of the intended argument type of the overridden method. The argument type replacability enables such a refinement. For instance on `String` we can use this replacability to replace the argument type of `equals` from `Object` to $\sim\text{String}$ in correspondence with the JavaDoc and the semantics of its implementation.

We can also gradually refine the argument. For instance `java.util.Calendar` similarly states “The result is `true` . . . only if the argument is a `Calendar` object. . .”, which in turn is further refined by “The result is `true` . . . only if the argument is a `GregorianCalendar` object. . .” as stated in the JavaDoc for the subclass `java.util.GregorianCalendar`. Using the replacability both classes can replace the argument types according to these specifications.

In the collection framework the replacability enables us to refine the `equals` method between interfaces, e.g. from `equals($\sim\text{Collection}\langle E \rangle$)` on `Collection` $\langle E \rangle$ to `equals($\sim\text{Set}\langle E \rangle$)` on `Set` $\langle E \rangle$ and even between interfaces and classes when refining the `equals` method to `equals($\sim\text{Set}\langle E \rangle$)` on `HashSet` $\langle E \rangle$, `TreeSet` $\langle E \rangle$, and so forth.

7 Experiments

We have implemented a full-scale Java 1.6 compiler which fully integrates related types into the type system. To test the added typing rules in a real world environment, the signatures of several methods in the standard library have been updated to use related types. A summary of the updated type annotations is shown in Table 1 where the notation $\forall M\langle K, V \rangle <: \text{Map}\langle K, V \rangle$ indicates that the given method signatures have been updated for all subtypes $M\langle K, V \rangle$ of $\text{Map}\langle K, V \rangle$. The table also includes a ‘#hits’ column in which the number of errors found in our experiments is shown for each annotation?.

Table 1. Updated type annotations

Host	Method	#hits
$\forall M\langle K, V \rangle <: \text{Map}\langle K, V \rangle$	boolean equals($\sim \text{Map}\langle K, V \rangle$)	0
	boolean containsKey($\sim K$)	5
	boolean containsValue($\sim V$)	2
	V get($\sim K$)	20
	V remove($\sim K$)	0
$\forall L\langle E \rangle <: \text{List}\langle E \rangle$	boolean equals($\sim \text{List}\langle E \rangle$)	4
	int indexOf($\sim E$)	0
	int lastIndexOf($\sim E$)	0
$\forall S\langle E \rangle <: \text{Set}\langle E \rangle$	boolean equals($\sim \text{Set}\langle E \rangle$)	0
$\forall C\langle E \rangle <: \text{Collection}\langle E \rangle$	boolean equals($\sim \text{Collection}\langle E \rangle$)	0
	boolean contains($\sim E$)	8
	boolean containsAll($\text{Collection}\langle ? \text{ extends } \sim E \rangle$)	0
	boolean remove($\sim E$)	1
	boolean removeAll($\text{Collection}\langle ? \text{ extends } \sim E \rangle$)	0
	boolean retainAll($\text{Collection}\langle ? \text{ extends } \sim E \rangle$)	0
$\forall T$	boolean equals($\sim T$)	42

Two different kinds of code have been tested. Work-in-progress code is tested to investigate the relation between immature code and unintended dead code bugs. The expectation is that these errors occur frequently while code is being written and that they are not easily detected after they have been introduced. Production code is tested to investigate how unintended dead code bugs persist in mature code. The expectation is that errors persist especially if they just have a degrading but not fatal consequence on the application.

We have categorized the errors found during our experiments by cause and effect. We have identified 3 different error causes shown in Table 2 and 8 effect categories shown in Table 3. Examples of some of these error causes and effects can be found in Appendix A.

Table 2. Error Causes

Conceptual Identity

The intended and actual argument types are conceptually identical but type-wise unrelated. For instance a `Token` class is (often) conceptually identical to the `String` that it holds. The correct implementation (often) only needs a single call to a getter-method or field access to obtain the `String` value.

Off-by-One

Similarly to Conceptual Identity, the correct implementation only needs a single call or field access to obtain the intended value. Here there is no other conceptual relation between the actual and intended argument types. This error is common a result of asymmetric dotted sequences, like `a.b.c.d.equals(a.b.c)`.

Conceptual Mismatch

The use of the actual argument type reveals a misunderstanding of its relation to the intended type. There is no direct link between the two types. For instance, testing for equality between a type declaration and a method declaration.

7.1 Work-in-Progress Code Tests

To test work-in-progress code we have run our compiler on 102 compilers written as part of the undergraduate compiler course [24] at Aarhus University. Two versions of the compilers have been compiled. ‘Preliminary’ consists of the preliminary source files handed in during the half year course and ‘Final’ consists of the updated source files handed in for the final deadline. The ‘Final’ code thus contains corrections of the ‘Preliminary’ code as well as features implemented after the preliminary hand-in deadlines. The compilers are built on a common skeleton provided at the beginning of the course and include generated files for the lexer/parser. The code written by the students themselves is therefore only one seventh of the total code base. The test results are shown in Table 4.

We found 65 errors and no warnings in the two versions of the 102 compilers and none of the errors were false positives, that is, all trivial call sites caused unintended dead code bugs. The compiler code involves intensive map lookups and AST navigation with long dotted sequences like

```
id.getAncestor(PTypeDecl.class).getName().getText().equals(id.getText())
```

We therefore expect many errors, and compared to the production code tests, described in Section 7.2, we find relatively many: 65 errors in 1662 KLOC vs. 14 errors in \approx 4300 KLOC. Contrary to our expectations, though, the number of errors didn’t decrease over time. As shown in Table 4(a), the number of errors is nearly constant between the preliminary and final hand-ins.

To investigate this further we have looked at the cause and effect of the errors correlated with their persistence. Errors found in both Preliminary and Final are considered “persistent”, whereas errors found only in Preliminary or Final are “removed” or “introduced”, respectively. What the ‘Total’ row of Table 4(b) shows is quite interesting: 15 errors are persistent throughout the preliminary

Table 3. Error Effects

Prevent Correctness

An error which makes the implementation incorrect. The effect of the incorrect implementation is not restricted to a singular domain as is the case for the remaining effect categories.

Prevent Optimization

An error which preserves the correctness of the program but makes the implementation less efficient or precise.

Prevent Cycle Detection

An error which prevents the intended detection of a cyclic structure, possibly resulting in infinite loops or stack overflow.

Prevent Error Detection

An error which prevents the application from detecting an actual error state.

False Error Detection

An error which makes the application wrongfully detect an error state.

Unused Code

An error residing in code which is never run by the current implementation.

Unintended Dead Code

An error residing in code which is dead due to another unintended dead code bug.

Null Pointer

An error which causes a `NullPointerException` in the following code. For instance the unchecked use of the returned value of a trivial call to `Map.get(~K)`.

and final versions, 18 errors have been removed before the final version, and 17 errors are introduced in the final version. In other words, it seems to be just as likely that an error is never found, as it is found and removed, or (re-)introduced during development.

As shown in Table 4(b) the simple error causes, Conceptual Identity and Off-by-One, account for half of the errors, but it is more interesting that Conceptual Mismatch account for the other half. The compilers are written while the students learn the concepts behind compilers, and the large number of conceptual mismatches shows just this; the lack of understanding of the concepts behind the code leads to the mixing of unrelated types. Another interesting observation regarding Conceptual Mismatch is the large number of removed and introduced errors, indicating that the students try to fix the errors, but lack the conceptual insight for doing so correctly.

The error effects shown in Table 4(c) are mainly prevention of application correctness with a few errors related to error detection in some form. Only a single error prevent optimization which, in relation to the findings in the production code shown in Section 7.2, can be explained by the code being work-in-progress: This is the students initial implementation so the focus is on functionality and correctness not on performance and precision.

Table 4. Work-in-progress code experimental results

(a) Results	Total KLOC	Student KLOC	Errors	Warnings
Preliminary	6154	723	33	0
Final	6369	939	32	0
Total	12523	1662	65	0
(b) Error causes	Persistent	Removed	Introduced	Total
Conceptual Identity	7	6	2	14
Off-by-One	3	3	4	11
Conceptual Mismatch	5	9	11	25
Total	15	18	17	50
(c) Error effects	Persistent	Removed	Introduced	Total
Prevent Correctness	12	10	6	28
Prevent Optimization	1			1
Prevent Cycle Detection			6	6
Prevent Error Detection		4	1	5
False Error Detection	1			1
Unused Code	1		1	2
Unintended Dead Code		1	1	2
Null Pointer		3	2	5

7.2 Production Code Tests

To test production code we have run our compiler on several versions of OpenJDK6 [20] and parts of the Eclipse [11] and Apache [1] projects. Since these projects are mature we do not expect many errors but as shown in Table 5 some errors are found.

Eight different builds of OpenJDK6 from build 05 of February 2008 to build 20 of June 2010 were tested, each consisting of approximately 2600 KLOC each. Through our annotations we found 6 errors and 4 warnings. Of these, none are false positives and 2 are new to FindBugs. All six errors and two of the warnings are persistent through all 8 builds. The last two warnings, those not found by FindBugs, are introduced in build 17 through an inconsistent introduction of new features (see Section A.3).

In the parts of Eclipse and Apache that were tested, 5 errors and 2 warnings are found. None of the five errors are false positives but the two warnings are. One of the warnings is a test case testing for `false` on a call to `equals`, i.e. the triviality was intended. The other warning is a case of testing objects of two seemingly unrelated interfaces for equality. Inspection of the implementing classes for these two interfaces reveals a single class which implements both interfaces, confirming our choice to make a distinction between errors and warnings in the S-RELATED subtyping rule in Figure 7.

Inspecting the identified error causes in Table 5(b), we see that most errors are either Conceptual Identity or Off-by-One which stem from simple deviations between the intended and actual code. As one might expect, Conceptual Mismatch

Table 5. Production code experimental results

(a) Results	KLOC	Errors	Warnings
OpenJDK6 (build 05-20)	≈ 2600	6	4
Apache projects	960	2	1
Eclipse IDE	710	3	1
(b) Error causes		Errors	Warnings
Conceptual Identity		5	1
Off-by-One		5	3
Conceptual Mismatch		1	0
(c) Error effects		Errors	Warnings
Prevent Correctness		3.5	1.5
Prevent Optimization		4.5	1.5
Prevent Error Detection		1	0
Unused Code		2	0
Unintended Dead Code		0	1

is more rare, indicating that the implementers have a deeper understanding of the concepts behind the code.

The identified error effects in Table 5(c) show a relatively large number of Prevent Optimization compared to the work-in-progress code. These errors are harder to detect since the implementation provides a correct result but only of an inferior quality, either as a less precise result, longer execution time or larger memory usage. It is therefore not surprising that such errors persist even in mature code, as is the case here.

8 Conclusion

We have shown that Java’s type system can be extended to capture a formal notion of relatedness among types and thus detect unintended dead code bugs. The required annotation burden is quite light and seems to capture invariants that are often already informally expressed in JavaDoc comments. For a full evaluation of the benefits of related types, though, we still need to do experiments to show to which extent this extended type system can reduce the time spent on debugging by flagging these errors at the time they are made.

Currently we do not check whether a method declared to be trivial on unrelated input is actually trivial. Future work lies in developing an analysis that will enable us to ensure this intended correlation.

The precision of our analysis seems good, but could actually be improved by an asymmetric version of relatedness. For instance expecting `List<Integer>` it is valid to provide an instance of `class NumList extends List<Number> { ... }`, since it could contain only `Integer` elements. On the other hand, expecting a `NumList` it makes no sense to provide a `List<Integer>` since it can never at runtime hold a `NumList` instance.

The method override rules are often used to emulate a self reference type, like `MyType` in LOOJ [3] or `This` in FGJ_{stc} [23]. In an extension of our type system including such a self reference type, `Self`, we could for instance express the signature of `equals` more directly as `equals(~Self)` and thus only explicitly refine the argument in cases like `equals(~List(E))` on `ArrayList(E)`.

References

1. The Apache Software Foundation. The Apache Software Foundation (2010), <http://www.apache.org/>
2. Bourrillion, K., et al.: Guava: Google Core Libraries for Java 1.5+ (2010), <http://code.google.com/p/guava-libraries/>
3. Bruce, K., Bruce, K.B., Foster, J.N.: LOOJ: Weaving LOOM into Java. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 390–414. Springer, Heidelberg (2004)
4. Bruce, K., Cardelli, L., Castagna, G., Leavens, G.T., Pierce, B.: On binary methods. *Theor. Pract. Object Syst.* 1, 221–242 (1995)
5. Cameron, N., Drossopoulou, S., Ernst, E.: A Model for Java Wildcards. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 2–26. Springer, Heidelberg (2008)
6. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.* 28, 517–575 (2006)
7. Colebourne, S.: Joda Primitives (2010), <http://joda-primitives.sourceforge.net/>
8. Copeland, T., et al.: PMD (2010), <http://pmd.sourceforge.net/>
9. Dautelle, J.M.: Javolution - The Java™ Solution for Real-Time and Embedded Systems (2010), <http://javolution.org/>
10. Dead Code Detector (DCD). Oracle Corporation (2010), <https://dcd.dev.java.net/>
11. Eclipse. The Eclipse Foundation (2010), <http://www.eclipse.org/>
12. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley Longman, Amsterdam (2005)
13. Hovemeyer, D., Pugh, W.: Finding bugs is easy. *ACM SIGPLAN Notices*, 132–136 (2004)
14. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 132–146 (1999)
15. Igarashi, A., Viroli, M.: On variance-based subtyping for parametric types. In: Deng, T. (ed.) ECOOP 2002. LNCS, vol. 2374, pp. 441–469. Springer, Heidelberg (2002)
16. Java™ Platform, Standard Edition 6 API Specification. Oracle Corporation (2010), <http://java.sun.com/javase/6/docs/api/>
17. Kennedy, A.J., Pierce, B.C.: On Decidability of Nominal Subtyping with Variance (2006); FOOL-WOOD 2007
18. Lindholm, T., Yellin, F.: Java Virtual Machine Specification, 2nd edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
19. NetBeans IDE. Oracle Corporation (2010), <http://netbeans.org/>
20. OpenJDK. Oracle Corporation (2010), <http://openjdk.java.net/>
21. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
22. Pugh, B., et al.: FindBugs™ - Find Bugs in Java Programs (August 2009), <http://findbugs.sourceforge.net/>
23. Saito, C., Igarashi, A.: Self type constructors. *SIGPLAN Not.* 44, 263–282 (2009)

24. Schwartzbach, M.I.: Design Choices in a Compiler Course or How to Make Undergraduates Love Formal Notation. In: CC, pp. 1–15 (2008)
25. Spieler, J.: Unnecessary Code Detector (2010), <http://www.ucdetector.org/>
26. Torgersen, M., Hansen, C.P., Ernst, E.: Adding wildcards to the Java programming language. *Journal of Object Technology*, 1289–1296 (2004)
27. Vigna, S.: fastutil: Fast & compact type-specific collections for Java™ (2010), <http://fastutil.dsi.unimi.it>

A Error Examples

In this section we present in detail a few of the errors found, taken from both the work-in-progress and production code and chosen to exemplify some of the different causes and effects of the unintended dead code bugs that our approach can detect.

A.1 Conceptual Identity: Lookup

A common error found in the code from the compilation course is shown below. In these compilers we use environment maps to provide lookup functionality for type members. For instance, each type declaration holds a field environment, mapping field names to field declarations. The keys are strings but from the AST the identifiers are provided through a `TIdentifier` token class. Conceptually the `TIdentifier` is the string that it holds, but implementation-wise it is not equal to the string. At the indicated line below, the field environment is checked for containment of the unrelated identifier `id`, thus preventing the correctness of the lookup method. Later in the same line we have another error in the attempt to return the mapped field declaration. This second error never occurs at runtime as a consequence of the first error; an effect we categorize as Unintended Dead Code in Table 3. The code `id` should in both cases correctly have read `id.getText()`.

```
public class PTypeDecl extends Node {
    public Map<String,AFieldDecl> field_env = ...
}
public class Disambiguation extends DepthFirstAdapter
    public Node lookupName(TIdentifier id) {
        PTypeDecl decl = findNode(id, PTypeDecl.class); ...
        if (decl.field_env.containsKey(id)) return decl.field_env.get(id); ...
    } ...
}
```

A.2 Conceptual Identity: Build Hierarchy

An example of an unintended dead code bug with a more subtle effect is shown in the code below. The code is from a compiler phase where the class hierarchy must be computed for each class. The students have wisely chosen to compute this hierarchy only once for each class, instead of the naïve approach of computing the hierarchy repeatedly. For this, they have chosen to store all previously computed class hierarchies in the `classHierarchy` map, using the name of the class as key.

```

public final class AClassTypeDecl extends PTypeDecl {
    public TIdentifier getName() { ... } ...
}
public class HierarchyCheck extends DepthFirstAdapter {
    public static Map<String,LinkedList<AClassTypeDecl>> classHierarchy = ...
    public static void buildHierarchy(AClassTypeDecl class_decl) {
        if ((classHierarchy.get(class_decl.getName()) == null)) {
            ...// build hierarchy for class_decl
        }
    } ...
}
}

```

At the indicated line in the code, this map is checked to see whether the class hierarchy has already been computed for the given `class_decl` or if it should be computed now. Unfortunately, the `class_decl.getName()` call returns the unrelated `TIdentifier` holding the name of the class, making this call-site trivial. As a consequence, the class hierarchy is computed repeatedly, and the intended optimization is therefore prevented. This error is, as one might expect, persistent. The implementation produces the correct result, so there is no obvious runtime behavior to flag the error. The code `class_decl.getName()` should correctly have read `class_decl.getName().getText()`.

A.3 Off-by-One: `ModelChannelMixer` vs. `SoftChannelMixerContainer`

In OpenJDK6 build 17 the class `com.sun.media.sound.SoftMainMixer` was refactored. Internal `ModelChannelMixer` objects are no longer stored directly but instead through an inner class `SoftChannelMixerContainer` which additionally holds buffer information for the mixer. This update is inconsistent in that not all references to `ModelChannelMixer` objects are correctly updated to an access through the `mixer` field of `SoftChannelMixerContainer`.

```

public class SoftMainMixer {
    private class SoftChannelMixerContainer { ModelChannelMixer mixer; ... }
    private Set<ModelChannelMixer> stoppedMixers = null;
    protected void processAudioBuffers() { ...
        SoftChannelMixerContainer[] act_registeredMixers; ...
        for (SoftChannelMixerContainer cmixer : act_registeredMixers) { ...
            if (stoppedMixers.contains(cmixer)) {
                stoppedMixers.remove(cmixer);
                cmixer.mixer.stop();
            } ...
        } ...
    } ...
}

```

At the indicated lines in the code above we have two Off-by-One dead code bugs, both generating *warnings* through the use of related types. In the first line the `stoppedMixers` set is checked for containment of the unrelated `cmixer` object and in the second line there is an attempt to remove the unrelated object from

the set. The effect of the error in the first line is that stopped mixers are never disposed but it is not clear whether this prevents correctness of the implementation or (just) prevents optimization, i.e. the freeing of system resources. The error in the second line is another example of the error effect Unintended Dead Code. Both error lines should correctly have used `cmixer.mixer` instead of `cmixer`.

The correct use of `cmixer.mixer` in the line following the errors indicates that the refactoring was performed by letting the type checker flag all uses of `mixer` objects that needed update, thus leaving the two error lines unaltered. The use of related types would have aided in such an approach to refactoring.

A.4 Off-by-One: Win32ShellFolder2

An interesting Off-by-One error was found in `sun.awt.shell.Win32ShellFolder2` of OpenJDK6. The use of `this` is subject to an “inner class capture”: The intended target of `this`, at the indicated line, is clearly the enclosing `Win32ShellFolder2` object, but since the code has been nested within an anonymous `ComTask<File[]>` class, `this` instead points to the inner class, preventing the correctness of the implementation in that the enclosing folder object is never recognized as the desktop folder. The code `this` should correctly have read `Win32ShellFolder2.this`.

```
final class Win32ShellFolder2 extends ShellFolder {
    public File[] listFiles(final boolean includeHiddenFiles) { ...
        return new ComTask<File[]>() {
            public File[] call() throws Exception { ...
                Win32ShellFolder2 desktop = ...
                if (this.equals(desktop) && ...) ...
            }
        }.execute();
    } ...
}
```

A.5 Conceptual Mismatch: ProjectDescription

An example of a Conceptual Mismatch error is found in the `ProjectDescription` class in the `org.eclipse.core.internal.resources` package in Eclipse. Here the class `java.net.URI` is tested against the Eclipse-specific interface `IPath` at the indicated line in the code below. The conceptual mismatch is that whereas `IPath` may be conceptually close to `URI` it *is not* a `URI` (nor does it even *have* a `URI`). The effect of this error is that the method always returns `true` and thus prevents the optimization this method was intended for.

```
import java.net.URI; ...
public class ProjectDescription extends ... {
    protected URI location = null;
    public boolean hasPrivateChanges(ProjectDescription description) { ...
        IPath otherLocation = description.getLocation(); ...
        return !location.equals(otherLocation);
    } ...
}
```

A.6 Conceptual Mismatch: Argument Types

Another case of a Conceptual Mismatch is found in the compiler code. Here, the formals of a set of constructors are checked against the invocation arguments in order to find the matching constructor. Using related types, a significant misunderstanding of the concepts behind the code is revealed through the unintended dead code bug found at the indicated line. Two `List<PLocalDecl>` and `List<PExp>` objects are compared, that is, a list of local declarations and a list of expressions, and since `PLocalDecl` and `PExp` are unrelated, so are these lists. What should have been compared is the type of the local declarations, found through `local_decl.getType()` and the type of the expressions, found through `exp.type`. The effect of the error is prevention of correctness in that only invocations with zero arguments are linked and for all other constructor invocations the targeted constructor is reported as not found.

```

public final class AConstructorDecl extends PDecl {
    List<PLocalDecl> getFormals() { ... } ...
}
public class TypeChecking extends DepthFirstAdapter {
    public void inASuperStm(ASuperStm stm) { ...
        for (PDecl member : superDecl.getMembers()) { ...
            if (member instanceof AConstructorDecl) {
                if (((AConstructorDecl)member).getFormals().equals(stm.getArgs()))
            {
                stm.constructor_decl = (AConstructorDecl)member; ...
            }
        }
    }
} ...
}

```

Gradual Typestate

Roger Wolff¹, Ronald Garcia^{1,*}, Éric Tanter^{2,**}, and Jonathan Aldrich^{1,***}

¹ School of Computer Science – Carnegie Mellon University

`first.last@cs.cmu.edu`

² PLEIAD Laboratory

Computer Science Department (DCC) – University of Chile

`etanter@dcc.uchile.cl`

Abstract. Typestate reflects how the legal operations on imperative objects can change at runtime as their internal state changes. A typestate checker can statically ensure, for instance, that an object method is only called when the object is in a state for which the operation is well-defined. Prior work has shown how modular typestate checking can be achieved thanks to access permissions and state guarantees. However, static typestate checking is still too rigid for some applications.

This paper formalizes a nominal object-oriented language with mutable state that integrates typestate change and typestate checking as primitive concepts. In addition to augmenting the types of object references with access permissions and state guarantees, the language extends the notion of *gradual typing* to account for typestate: gradual typestate checking seamlessly combines static and dynamic checking by automatically inserting runtime checks into programs. A novel flow-sensitive permission-based type system allows programmers to write safe code even when the static type checker can only partly verify it.

1 Introduction

This paper investigates an approach to increasing the expressiveness and flexibility of object-oriented languages as a means to improve the reliability of software. By introducing *typestate* directly into the language and extending its type system with support for *gradual typing*, useful abstractions can be implemented directly, stronger program properties can be enforced statically, and when necessary dynamic checks can be introduced seamlessly.

An object’s type specifies the methods that can be called on it. In most programming languages, this type is constant throughout the object’s lifetime, but in practice, the methods that it makes sense to call on an object change as its

* Supported by the National Science Foundation under Grant #0937060 to the Computing Research Association for the CIFellows Project.

** Partially funded by FONDECYT project 1110051.

*** Support from grants NSF #CCF-0811592, DARPA #HR00110710019, ARO #DAAD19-02-1-0389 to CyLab, and CMU|Portugal Aeminium #CMU-PT/SE/0038/2008.

runtime state changes (e.g., an open file cannot be opened again). These constraints typically lie outside the reach of standard type systems, and unintended uses of objects result, at best, in runtime exceptions.

More broadly, types generally denote properties that hold without change, and in mainstream type systems, they fail to account for how changes to mutable state can affect the properties of an object. To address this shortcoming, Strom and Yemini [26] introduced the notion of *typestate* as an extension of the traditional notion of type. Typestate reflects how the legal operations on imperative objects can change at runtime as their internal state changes.

The seminal work on typestate [26] focused primarily on whether variables were properly initialized, and presented a static *typestate checker*. A typestate checker must account for the flow of data and control in a program to ensure that objects are used in accordance with their state at any given point in a computation. Since that original work, typestate has been used to codify and check more sophisticated state-dependent properties of object-oriented programs. It has been used, for instance, to verify object invariants on .NET [10], to verify that Java programs adhere to object protocols [12, 5, 7], and to check that groups of objects collaborate with each other according to an interaction specification [20].

Most imperative languages cannot express typestates directly: rather, typestates are encoded through a disciplined use of member variables. For instance, consider a typical object-oriented file abstraction. A closed file may have a `null` value in its file descriptor field. Accordingly, the `close` method of the file object first checks if the file descriptor is `null`, in which case it throws an exception to signal that the file is already closed. Such typestate encodings hinder program comprehension and correctness. Comprehension is hampered because the protocols underlying the typestate properties, which reflect a programmer’s intent, are at best described in the documentation of the code. Also, typestate encodings cannot guarantee by construction that a program does not perform illegal operations. Checking typestate encodings can be done through a whole-program analysis (e.g. [12]), or with a modular checker based on additional program annotations (e.g. [4]). In either case, the lack of integration with the programming language hinders adoption by programmers.

To overcome the shortcomings of typestate encodings, a typestate-oriented programming (TSOP) language directly supports expressing them [2]. For instance, in a class-based language that supports dynamically changing an object’s class (like Smalltalk and its `become` statement), typestates can be represented as classes and be dynamically updated: objects can have typestate-dependent interfaces, behaviors, and representations. Protocol violations in a dynamically-typed TSOP language however result in “method not found” errors. To avoid such errors, it is crucial to regain the guarantees provided by static type checking. Static typestate checking is challenging, especially in the presence of aliasing. Some approaches sacrifice modularity and rely on whole program analyses [12, 20, 7]; others retain modularity at the expense of sophisticated type systems, typically based on linear logic [27] and requiring many annotations.

One kind of annotations is *access permissions*, which specify certain aliasing patterns [8, 10, 4]. Unfortunately, these systems cannot always verify safe code, due to the conservative assumptions they must make. Advanced techniques like fractional permissions [8] increase the expressiveness of a type system, within limits, but increase its complexity.

Many practical languages already provide a simple feature for overcoming the limitations of their type systems: dynamic coercions. Although these coercions (a.k.a. casts) may fail at runtime, they are often necessary in specific scenarios where the static machinery is insufficient. Runtime assertions related to typestates are not supported by any modular approach we know of; one primary objective of this work is to support them.

In addition, once dynamic coercions on typestates are available, they can be used to ease the transition from dynamically- to statically-typed code. We therefore extend gradual typing [25, 24] to account for typestates: we make typestate annotations optional, check as much as possible statically, and automatically insert runtime checks into programs where needed. This allows programmers to gradually annotate their code and get progressively more support from the type checker, while still being able to safely run a partially-annotated program.

The primary contribution of this work is Gradual Featherweight Typestate (GFT), a core calculus for typestate-oriented programming inspired by Featherweight Java [17], which supports dynamic permission checking and gradual typing. The proposed language addresses the most important issues of current typestate checkers. GFT directly integrates typestate as a first-class language concept. Its analysis is modular and safe without imposing complex notions like fractional permissions onto programmers. It also supports recovery of precise typing using dynamically-checked assertions and supports the gradual addition of type annotations to a program.

Section 2 introduces the key elements of typestate-oriented programming with access permissions and state guarantees. Section 3 describes the static subset of GFT, and Section 4 presents the extensions for dynamic permission checking and gradual typing. The semantics are presented using a type-safe internal language (Section 5) to which GFT translates (Section 6). The soundness proof is available in a companion technical report [28]. Section 7 concludes. A translator for the source language, type checker for the internal language, and executable runtime semantics are available at:

<http://www.cs.cmu.edu/~rxg/gft.html>.

2 Typestate-Oriented Programming

In order to avoid conditionals on flag fields or other indirect mechanisms like the State pattern [13], typestate-oriented programming proposes to extend object-oriented programming with an explicit notion of *state* (from here on we use state to mean typestate). In TSOP, objects are modeled not just in terms of classes, but in terms of changing states. Each state may have its own representation and methods, which may transition the object to new states.

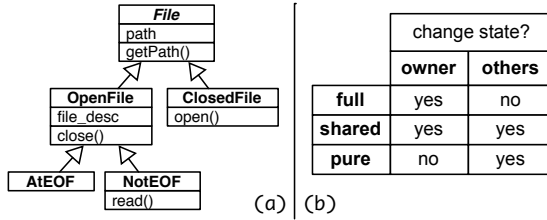


Fig. 1. (a) Hierarchy of files states. (b) Access permissions.

To illustrate this concept in practice, consider a familiar example. A file object has methods such as `open`, `close` and `read`. However, these methods cannot be called at just any time. A file can only be `read` after it has been `opened`; even if open, if we have reached the end-of-file, then `reading` is not available anymore; an open file cannot be opened again, etc. Figure 1a depicts a model of the file example in terms of states, using distinct classes in a subclass hierarchy to represent them. `File` is an abstract state; a file object is either in the `OpenFile` or `ClosedFile` state. Note that the `path` field is present in both states, but that the `file_desc` field, which refers to the low-level operating system resource, is only present in the `OpenFile` state. Any `OpenFile` can be closed; however, it is only possible to read from an open file if the end-of-file has not been reached. Therefore, the `OpenFile` state has two refining substates, `AtEOF` and `NotEOF`.

State change. A TSOP language supports a state change operation, denoted here by `<-`. For instance, the `close` method in `OpenFile` can be defined as:

```
void close() { this <- ClosedFile(this.path); }
```

The expression form `e <- S(...)` transitions the object described by `e` into the state `S`; the arguments are used to initialize the fields of the object. In other words, `<-` behaves like a constructor, but updates the object in-place.

Declaring state changes. A statically-typed TSOP language must track state changes in order to reject programs that invoke methods on objects in inappropriate states. Consider the following:

```
OpenFile f = ...; f.close(); f.close();
```

The type of `f` before the first call to `close` is `OpenFile`. However, the second call to `close` should be rejected by a typechecker. One way to do so is to analyze the body of the `close` method to deduce that it updates the state of its argument to `ClosedFile`. However, this approach sacrifices modularity. Therefore, a method’s signature should specify the output state of its arguments as well as that of its receiver. The calculi in this paper specify the state changes of methods by annotating each argument with its input and output state, separated by the `>>` symbol. The input and output states of the receiver object are placed in square brackets after the normal argument list, e.g.:

```
void close() [OpenFile >> ClosedFile] {...}
```

Access permissions. In a language with aliasing, tracking state changes is a subtle process. For instance, consider the following (where `F`, `OF` and `CF` are abbreviations for `File`, `OpenFile` and `ClosedFile`, respectively):

```
void m(OF >> CF f, OF >> OF g) {f.close(); print(g.file_desc.pos);}
```

Because of possible aliasing, `f` and `g` may refer to the same object. In that case, the method body of `m` must not be well-typed, as `g` may refer to a closed file by the time it needs to access its (potentially non-existent) `file_desc` field.

To track state changes in the presence of aliasing, Bierhoff and Aldrich have proposed *access permissions* [4, 5]. An access permission specifies whether a given reference to an object can be used to change its state or not, as well as the access permissions that other aliases to the same object might have. In this work we consider three kinds of access permissions (Figure 10): `full`, `shared` and `pure`. We say a reference has *write access* if it has the ability to change the state of an object. `full` and `shared` have write access, where `full` implies *exclusive* write access.

One fix for the `m` method is to require that `f` and `g` have exclusive write access to an `OF` in order to ensure that they are not aliases, and therefore that `f.close()` cannot affect `g`'s referent.

```
void m(full OF >> full CF f, full OF >> full OF g){ ... }
```

State guarantees. Requiring `g` to have exclusive write access seems like overkill here. Only a `pure` permission is required to read the field `file_desc`. But we must still ensure that the two parameters are not aliases.

For more flexible reasoning in the presence of aliasing, access permissions are augmented with *state guarantees* (proposed by Bierhoff and Aldrich [4] but formalized and proven sound for the first time here). A state guarantee puts an upper bound on the state change that may be performed by a reference with write access: it can only transition an object to some subclass of the state guarantee. A type specification then has the form `k(D) C` where `k` is the access permission, `D` is the state guarantee, and `C` is the current state of the object. A *permission*, `k(D)`, is the access permission coupled with the state guarantee [4].

Consider:

```
full(Object) NotEOF x = new NotEOF(...);
pure(OF) OF y = x;
x.read();
print(y.file_desc.pos);
```

While `x.read()` may change the state of the file by transitioning it to `AtEOF`, it cannot invalidate the open file assumption held by `y`.

State guarantees improve modular reasoning about typestates substantially. For instance, they recover the ability to express something similar to an ordinary object-oriented type: `shared(C) C` allows an object to be updated but guarantees that it will always obey the interface `C`. Also, it turns out that we can use

¹ When it is clear from the context, we sometimes say 'permission' when we really mean 'access permission'.

state guarantees to express an alternative solution to the previous example: restrict `g` to the `pure` access permission it requires, but add a state guarantee of `OF` to ensure that no other reference can transition the object to `ClosedFile`:

```
void m(full(F) OF >> full(F) CF f,
      pure(OF) OF >> pure(OF) OF g){ ... }
```

In this case, we can still statically enforce that `f` and `g` are not aliases by carefully choosing exactly how references to objects can be created. In this way, we can allow the programmer more flexibility than always demanding exclusive access to objects.

Permission flows. Permissions are split between all aliases and carefully restricted to ensure safety. This includes aliases in local variables, as well as in object fields. Consider the following snippet:

```
class FileContainer{ shared(OF) OF file; }

full(Object) OF x = new OF(...);
pure(OF) OF y = x;
full(Object) FileContainer z = new FileContainer(x);
```

After construction of the `OF`, the reference `x` has no aliases, so it is safe to give it full permission with an unrestricted update capability (`Object` state guarantee). Then, a local alias `y` is created, capturing a `pure` permission with `OF` guarantee. After this point, any state change done through `x` must respect this guarantee. Therefore, the permission of `x` must be downgraded to `full(OF)`. Finally, a container object is created, passing `x` as argument to the constructor. The field of `z` captures a `shared(OF)` permission. The permission of `x` is downgraded again, this time to `shared(OF)`. At this point, there are three aliases to the same file object: `x` and `z.file` both hold a `shared(OF)` permission, and `y` holds a `pure(OF)`. All aliases must be consistent, in that a state update through one alias must not break the invariants of other references.

Temporarily holding permissions In general, as the program executes, permissions to variables get split and are strictly weakened. There are many ways to refine the static type system in order to increase expressiveness, such as parametric polymorphism, fractional permissions and borrowing [8, 9]. Here we consider one such refinement: a mechanism that can temporarily hold some permissions to a reference while a sub-computation is performed. Consider the following:

```
void printPath(pure(F) F >> pure(F) F);

full(Object) OF x = new OF(...);
printPath(x);
x.close();
```

This program is ill-typed due to the downgrading of permissions. In order to invoke `printPath`, the permission to `x` is downgraded from `full(Object) OF` to `pure(F) F`. Therefore, `close`, which requires a `full(F) OF`, cannot be called,

although the call to `close` is safe: `printPath` requires a read-only alias to its argument, and there are no other writeable aliases to `x`. This is an unfortunate limitation due to the conservative nature of the type system.

A workaround is to introduce a temporary alias to `x` with only a `pure` permission, and use that alias to invoke `printPath`. This is however cumbersome and does not allow for permissions to be merged back later on. In order to properly support this pattern, we introduce a novel expression, `hold`, which reserves a permission to a variable for use within a lexical scope, and then *merges* that permission with that of the variable at the end of the scope. For instance:

```
full(Object) OF x = new OF(...);
hold[x:full(F) OF] { printPath(x); }
x.close();
```

Dynamic asserts. As sophisticated as the type system might be (supporting `hold`, borrowing, etc.), it is still necessarily conservative and therefore loses precision. Dynamic checks, like runtime casts, are often useful to recover such precision. For instance, consider the following extension of the `FileContainer` snippet seen previously in which both `y` and `z` are updated to release their aliases to `x`.

```
...
y = new OF(...);
z <- Object();
assert<full(F) OF> x;
x.close();
```

Assuming `close` requires a `full(F)` permission to its receiver, the type system is unable to determine that `x` can be closed, even though it is safe to do so (`x` is once again the sole reference to the object). A *dynamic assert* allows this permission to be recovered. Like casts, dynamic asserts may fail at runtime.

Gradual typing. A statically typed TSOP program requires more annotations than a comparable OO program. This may be prohibitively burdensome for a programmer, especially during the initial stages of development. For this reason, we develop a gradually typed calculus that supports a dynamic type `Dyn`. Precise type annotations can then be omitted from an early draft of a program as in the following code:

```
Dyn f = ...; f.read();
```

A runtime check will verify that `f` refers to an object that has a `read` method². Assume that `read` is annotated with a receiver type `full(OF) NotEOF`. In this case, we must ensure that we have an adequate permission to the receiver. Thus, a further runtime check will verify that `f` refers to an object that is currently in the `NotEOF` state, that no aliases have write access, and that all aliases have a state guarantee that is a superstate of `OF`. The last two conditions ensure that invariants of aliases to `f` cannot be broken. Gradual typing thus enables dynamically and statically-typed parts of a program to coexist without compromising safety.

² Note that `Dyn` is different from `Object`; if `f` had type `Object` the typechecker would check for a method `read` in `Object`, and would raise an error.

Putting it all together. Listing 1 exhibits the above capabilities in a small logging example that generalizes to other shared resources³. The `OpenFileLogger` (`OFL`) state holds a reference to a file object (`OF`) and presents a `log` method for logging messages to it. When logging is complete, the `close` method acquires all permissions to the file by swapping in a sentinel value (with `:=:`, explained in the next section), closes the file, and transitions the logger to the `FileLogger` (`FL`) state, which has no file handle. The client code declares and uses two logging interfaces, `staticLog` and `dynamicLog`. They are somewhat contrived, but are meant to represent APIs that utilize a file logger but do not store it. After creating `logger` (line 17), the `file` reference no longer has enough permission to close the file, so calls to `logger.log` are safe. Line 19 holds a shared permission to `logger` during a dynamically-typed method call. By line 22, `logger` only has shared permission, though no other aliases exist. After asserting back full permission, `logger` can close the file log.

```

1  class FileLogger { /* Logging-related Data and Methods */
2  class OpenFileLogger : FileLogger {
3      full(OF) OF file;
4
5      Void log(string s)[shared(OFL) OFL >> shared(OFL) OFL] {...}
6      Void close()[full(FL) OFL >> full(FL) FL] {
7          full(OF) fileT = (this.file :=: new File("/dev/null"));
8          fileT.close();
9          this <- FileLogger();
10     }
11 }
12 // Client Code
13 Void staticLog(shared(OFL) logger >> shared(OFL) logger) { logger.log("in staticLog"); }
14 Dyn dynamicLog(Dyn logger) { logger.log("in dynamicLog"); }
15
16 full(OF) OF file = new OF(...);
17 full(OFL) OFL logger = new OFL(file);
18
19 hold[logger:shared(OFL) OFL]{ dynamicLog(logger); }
20 staticLog(logger);
21
22 assert<full(FL) OFL>(logger);
23 logger.close();

```

Listing 1. Sample Typestate-Oriented code

3 Static Featherweight Typestate

We present a formal model for a language with integrated support for gradual typestate. The language is inspired by Featherweight Java (FJ) [17], so we call it Gradual Featherweight Typestate (GFT). Gradually-typed languages are typically presented as two distinct languages: a fully static language and its gradual extension. We only describe the gradual language; this section focuses on its static aspects. Sections 4 and beyond present the extensions for gradual typing. Garcia et al. [14] formalizes a fully static variant of GFT, called Featherweight Typestate. The static subset of the language is novel in its own right: it is the

³ A practical language would provide means to abbreviate our type annotations.

$x, \text{this} \in \text{IDENTIFIERNAMES}$	
$m \in \text{METHODNAMES}$	
$f \in \text{FIELDNAMES}$	
$C, D, E, \text{Object} \in \text{CLASSNAMES}$	
$PG ::= \overline{CL}, e$	(programs)
$CL ::= \text{class } C \text{ extends } D \{ \overline{FM} \}$	(classes)
$F ::= T f$	(fields)
$M ::= T m(\overline{T} \gg \overline{T} x) [T \gg T] \{ \text{return } e; \}$	(methods)
$T ::= P C \mid \text{Void}$	(types)
$P ::= k(D)$	(permissions)
$k ::= \text{full} \mid \text{shared} \mid \text{pure}$	(access permissions)
$e ::= x \mid \text{let } x : T = e \text{ in } e \mid \text{let } x = e \text{ in } e \mid \text{new } C(\overline{x})$	(expressions)
$\quad \mid x.f \mid x.m(\overline{x}) \mid x.f ::= x \mid x \leftarrow C(\overline{x})$	
$\Delta ::= x : \overline{T}$	(type contexts)

Fig. 2. Gradual Featherweight Typestate: Syntax (static subset)

first formalization of a nominal TSOP language, with support for representing typestates as classes, modular typestate-checking and state guarantees, and an algorithmic flow-sensitive type system specification.

3.1 Syntax

Figure 2 presents GFT’s syntax. Smallcaps (e.g. FIELDNAMES) indicate syntactic categories, italics (e.g. C) indicate metavariables, and sans serif (e.g. **Object**) indicates particular elements of a category. Overbars (e.g. \overline{A}) indicate possibly empty sequences (e.g. A_1, \dots, A_n). GFT assumes a number of primitive notions, such as identifiers (including **this**) and method, field, and class names (including **Object**). A GFT program PG is a list of class declarations \overline{CL} paired with an expression e . Class definitions are standard, except that a GFT class does not have an explicit constructor: instead, it has an implicit constructor that assigns an initial value to each field. Fields F and methods M are standard. Each method parameter is annotated with its input and output types, and the method itself carries an annotation (in square brackets) for the receiver object. We use helper functions like *fields*, *method*, etc., whose definitions are omitted for brevity.

Types in GFT extend the Java notion of class names as types. As explained in Section 2, the type of a GFT object reference has two components, its permission and its class (or *state*). The permission can be broken down further into its access permission k and state guarantee D . We write these *object reference types* in the form $k(D) C$. The **Void** type classifies expressions executed purely for their effects. No source-level values have the **Void** type.

To simplify the description of the type system, expressions in GFT are restricted to A-normal form [22], so let expressions explicitly sequence all complex operations (we write $e_1; e_2$ as shorthand). An optional type ascription provides fine-grained control over how permissions are distributed to the bound variable (Section 3.3).

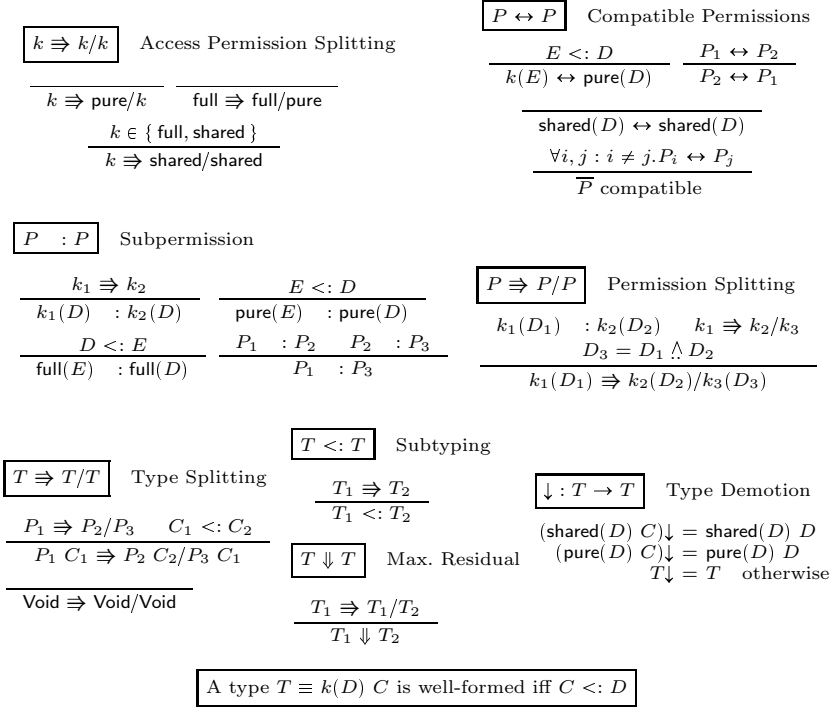


Fig. 3. Permission and Type Management Relations

Apart from method invocation, field reference and object creation (all standard), GFT includes the update operation $x_0 \leftarrow C(\overline{x_1})$ in support of typestate. It replaces the value of x_0 with the new object of class C , which may not be the same as x_0 's current class. Also non-standard is the swapping assignment $x_0.f := x_1$. It assigns the value of x_1 to the field f and returns the old value as its result. The need for this expression is detailed in Section 3.3.

3.2 Managing Permissions

Before we present typing judgments for GFT, we must explain how permissions are treated. Permissions to an object are a resource that is split among the variables and fields that reference it. Figure 3 presents several auxiliary judgments that specify how permissions may be safely split, and their relation to typing.

First, *access permission splitting* $k_1 \ni k_2/k_3$ describes how given a k_1 permission, permission k_2 can be acquired, leaving behind k_3 as the residual. When we are only concerned that a permission k_2 can be split from a permission k_1 (i.e. the residual permission is irrelevant), we write $k_1 \ni k_2$. For instance, given any permission k , $\text{full} \ni k$ and $k \ni k$.

Permissions partially determine what operations are possible, as well as when an object can be safely bound to an identifier. The restrictions on permissions

are formalized as a partial order on permissions, analogous to subtyping. The notation $P_1 <: P_2$ says that P_1 is a *subpermission* of P_2 , which means that a reference with P_1 permission may be used wherever an object reference with P_2 permission is needed. As expected, the subpermission relation is reflexive and transitive. Splitting an access permission produces a lesser (or identical) permission. The rules that mention *pure* and *full* capture how state guarantees affect the strength of permissions. Pure permissions covary with their state guarantee because a pure reference with a superclass state guarantee assumes less reading capability. Full permissions contravary with their state guarantee because a full reference with a subclass state guarantee assumes less writing capability (it can update to fewer possible states).

Permission splitting extends access permission splitting by accounting for state guarantees. First, if $k_1(D_1) <: k_2(D_2)$, splitting is safe. The question then is to determine the proper residual permission $k_3(D_3)$. The k_3 residual is obtained by splitting k_2 from k_1 . The resulting state guarantee D_3 is the greatest lower bound of D_1 and D_2 in the subclass hierarchy, denoted $D_1 \wedge D_2$ ⁴.

Permission splitting in turn extends to *type splitting* $T \Rightarrow T/T$, taking subclasses into account for object references; the *Void* type can be arbitrarily split. We use type splitting to define the notion of subtyping $T <: T$ used in GFT. As with base permission splitting, we write $P_1 \Rightarrow P_2$ or $T_1 \Rightarrow T_2$ to express that P_2 or T_2 can be split from P_1 or T_1 respectively.

The *maximum residual* relation $T_1 \Downarrow T_2$ specializes type splitting for the case where all the permissions to an object are acquired. The result type T_2 is what is leftover; for instance, $\text{full}(D) C \Downarrow \text{pure}(D) C$ and $\text{shared}(D) C \Downarrow \text{shared}(D) C$.

The *compatible permissions* relation $P_1 \leftrightarrow P_2$ says that two distinct references to the same object, one with permissions P_1 and the other with P_2 can soundly coexist at runtime. For instance, $\text{shared}(C) \leftrightarrow \text{shared}(C)$, and $\text{full}(C) \leftrightarrow \text{pure}(\text{Object})$. This notion is used to define the relation \overline{P} *compatible*: that the outstanding permissions \overline{P} of references to a particular object can all coexist.

Finally, we defer the discussion of *type demotion* to the end of Section 3.3.

3.3 Static Semantics

Armed with the permission management relations, we now discuss the salient features of the static semantics of GFT: flow-sensitive, deterministic typing through bidirectional type checking.

Flow-sensitive typing As with FJ, the GFT type system relies upon type contexts Δ . Whereas Γ is the standard metavariable for type contexts, we use a different metavariable Δ to emphasize that the typing contexts are not merely lexical: they are *linear* [16]. In GFT's type system, the types of identifiers are flow-sensitive in the sense that they vary over the course of a program. In part this reflects how the permissions to a particular object may be partitioned and shared between references as computation proceeds, but it also reflects how update operations may change the class of an object during execution.

⁴ $k_1(D_1) <: k_2(D_2)$ implies that D_1 and D_2 are related by subclassing

The GFT typing judgments are quaternary relations roughly of the form $\Delta_1 \vdash e : T \dashv \Delta_2$: given the typing assumptions Δ_1 , the expression e can be assigned the type T and produces typing assumptions Δ_2 as its output. The assumptions in question are the types of each reference. Threading typing contexts through the typing judgment captures the flow-sensitivity of type assumptions.

Deterministic type checking The type system specification must be elaborated to both ensure determinism of our type system and also retain flexibility. Consider a candidate typing judgment for variable references.

$$\frac{T_1 \Rightarrow T_2/T_3}{\Delta, x : T_1 \vdash x : T_2 \dashv \Delta, x : T_3}$$

It states that if x is assumed to have type T_1 , and T_1 can be split into T_2 and T_3 , then the expression x can be typed at T_2 . Because T_2 may not be unique, a source program may be well-typed according to multiple derivations, with each derivation representing a different split of permissions between this particular variable reference and later references to x . Nondeterminism is incompatible with dynamic permission assertions⁵: a system could succeed sometimes and fail other times if permissions could flow more than one way for the same code.

Rather than requiring type annotations for all variable references, we use bidirectional typing [21] to define a deterministic type system in which annotations are used only to tune how permissions are split.

The type system is structured as two mutually recursive judgments. The *type synthesis* judgment $\Delta_1 \vdash e \Rightarrow T \dashv \Delta_2$ conceptually analyzes the expression e in the context Δ_1 and synthesizes a type T for it; the type T is an output of the judgment, along with the output context Δ_2 . The *type checking* judgment $\Delta_1 \vdash e \Leftarrow T \dashv \Delta_2$ checks that the expression e under the type context Δ_1 can be given the type T . The type T is an input to the judgment, and the only output is the context Δ_2 .

Typing rules Figure 4 presents some of the typing rules for GFT expressions⁶. A variable reference is typed differently depending on whether its type is synthesized or checked. The synthesis rule ($\text{ctx} \Rightarrow$) yields maximal permissions to the referenced object. Its output context associates the maximum residual permissions to the variable. In contrast, the checking rule ($\text{ctx} \Leftarrow$) just ensures that the desired type can be split from the starting type, and leaves the corresponding residual in the output context.

Each of the typing rules for let represents both a checking and synthesis rule. Replacing the \Leftrightarrow with \Leftarrow or \Rightarrow gives the checking or synthesis rule, respectively. The ($\text{let} \Leftarrow$) and ($\text{let} \Rightarrow$) rules differ only in how the bound expression is typed. When the bound variable has a type ascription, $x : T_1$, the expression e_1 is checked against that type. If there is no type ascription, e_1 's type is synthesized.

⁵ Determinism is not important for the fully static case: Featherweight Typestate uses non-deterministic typing rules [14].

⁶ As with FJ, well-typing is extended to apply to whole programs. The details are covered in the technical report [28].

$$\begin{array}{c}
(\text{ctx}\Rightarrow) \frac{T_1 \Downarrow T_2}{\Delta, x : T_1 \vdash x \Rightarrow T_1 \dashv \Delta, x : T_2} \qquad (\text{ctx}\Leftarrow) \frac{T_1 \Rightarrow T_2/T_3}{\Delta, x : T_1 \vdash x \Leftarrow T_2 \dashv \Delta, x : T_3} \\
(\hat{e}\Leftarrow) \frac{\Delta \vdash \hat{e} \Rightarrow T_2 \dashv \Delta_1 \quad T_2 <: T_1}{\Delta \vdash \hat{e} \Leftarrow T_1 \dashv \Delta_1} \\
(\text{let}\Leftarrow) \frac{\Delta \vdash e_1 \Rightarrow T_1 \dashv \Delta_1 \quad \Delta_1, x : T_1 \vdash e_2 \Leftarrow T_2 \dashv \Delta_2, x : T}{\Delta \vdash \text{let } x = e_1 \text{ in } e_2 \Leftarrow T_2 \dashv \Delta_2} \\
(\text{letT}\Leftarrow) \frac{\Delta \vdash e_1 \Leftarrow T_1 \dashv \Delta_1 \quad \Delta_1, x : T_1 \vdash e_2 \Leftarrow T_2 \dashv \Delta_2, x : T}{\Delta \vdash \text{let } x : T_1 = e_1 \text{ in } e_2 \Leftarrow T_2 \dashv \Delta_2} \\
(\text{update}\Rightarrow) \frac{\begin{array}{c} \text{fields}(C_2) = \overline{T_2} f \quad \Delta \vdash \overline{x_2} \Leftarrow \overline{T_2} \dashv \Delta', x_1 : k(D) C \\ k \in \{ \text{full}, \text{shared} \} \qquad C_2 <: D \end{array}}{\Delta \vdash x_1 \leftarrow C_2(\overline{x_2}) \Rightarrow \text{Void} \dashv \Delta' \downarrow, x_1 : k(D) C_2} \\
(\text{ref}\Rightarrow) \frac{T_2 f \in \text{fields}(C_1) \quad T_2 \Downarrow T'_2}{\Delta, x : P_1 C_1 \vdash x.f \Rightarrow T'_2 \dashv \Delta, x : P_1 C_1} \\
(\text{swap}\Rightarrow) \frac{T_2 f \in \text{fields}(C_1) \quad \Delta, x_1 : P_1 C_1 \vdash x_2 \Leftarrow T_2 \dashv \Delta'}{\Delta, x_1 : P_1 C_1 \vdash x_1.f := x_2 \Rightarrow T_2 \dashv \Delta'}
\end{array}$$

Fig. 4. Select Expression Typing Rules

The typing rules for let and variable references combine to determine how permissions transfer between references. When a variable reference is bound to another variable, the new variable by default acquires maximal permissions to the referenced object; A type annotation on the let-bound variable can tune how permissions are transferred to a binding. For instance, assume x has type $\text{full}(D) C$ and consider the two expressions:

- (1) $\text{let } y = x \text{ in } e$
- (2) $\text{let } y : \text{shared}(D) C = x \text{ in } e$

In expression (1) y has $\text{full}(D) C$ type and x has $\text{pure}(D) C$ type in e , but in expression (2) both x and y have $\text{shared}(D) C$ type.

Type checking is treated uniformly for all other expressions. The $(\hat{e} \Leftarrow)$ rule schematically expresses checking for those expressions, which we indicate with \hat{e} . For all of them, type checking can be characterized simply in terms of type synthesis: an expression checks at type T_1 if its type synthesizes to some subtype T_2 of T_1 . The rest of the expressions in the language only require type synthesis rules (see Section [6](#)).

A variable reference can only perform an update if it has write access. The arguments to the constructor—which can include the reference being updated—are type checked at the target class’s field types. We use the shorthand notation $\Delta \vdash x \Leftarrow \overline{T} \dashv \Delta'$ to stand for iteratively checking the arguments:

$$\Delta = \Delta_0 \vdash x_0 \Leftarrow T_0 \dashv \Delta_1; \quad \dots \quad ; \Delta_n \vdash x_n \Leftarrow T_n \dashv \Delta_{n+1} = \Delta'$$

The target class of the update must respect the updated reference’s state guarantee, taking into account any uses of that reference in the construction of the new object. The update operation is performed solely for its effect on the heap,

so the type of the overall expression is `Void`. The output type of the updated object reflects its new class.

Type demotion. Update operations can alter the state of any number of variable references. To retain soundness in the face of these operations, it is sometimes necessary to discard previously known information in case it has been invalidated. In these cases, an object reference’s class must revert to its state guarantee, which is a trusted state after an update. The *type demotion* function $T\downarrow$ (Figure 3) expresses this restricting of assumptions. Note that full references need not be demoted since no other reference could have changed their states. We write $\Delta\downarrow$ for the compatible extension of demotion to typing contexts.

The synthesis rule for the update operation in Figure 4 makes use of type demotion: type assumptions from the input context are demoted in the output context to ensure that any aliases to the updated object retain a conservative approximation of the object’s current class⁷.

Note that type demotion does not imply any runtime overhead: it is a purely static process. Furthermore, types of class fields have the restriction that they must be invariant under demotion (i.e. $T\downarrow = T$). Since the types of fields do not change as a program runs, they must not be invalidated by update operations. This restriction ensures that field types remain compatible with other aliases to their objects. Also, as a result only local variable types need ever be demoted.

Field Operations. As was mentioned in Section 3.1, two operations operate directly on the an object field: field reference and swapping assignment. Their type synthesis rules appear in Figure 4. Field reference operations do not relinquish any of the permissions held by a field, so the result type is determined by taking the maximal residual T'_2 of the field type T_2 . This operation does not affect the permissions of the object reference used to access the field.

Swap operations cause an object to relinquish all permissions to a field and replace it with a new reference. The swap expression has two purposes. The first is to reassign a field value in the heap. The second is to return the old field value as the result of the expression. If a field has `shared` or `pure` permissions to an object, then field reference can yield the same amount of permission; however, if a field has `full` permission to an object, only swapping can yield a full permission.

3.4 Holding Permissions

The static fragment of GFT described above captures the essence of a TSOP language, but the design can be usefully extended. For instance, Section 2 introduced the hold expression $\text{hold}[x : T](e)$, which captures the amount of x ’s permissions denoted by T for the duration of the computation e . When e completes, these permissions are merged back into x .

$$e ::= \dots \mid \text{hold}[x : T](e)$$

⁷ The language could retain more precise types by demoting only objects with types related to the updated object. For simplicity of presentation, we demote uniformly.

The hold expression is a useful but orthogonal addition to the type system. A practical TSOP language would build hold’s capabilities directly into the method invocation mechanism so as to preserve permissions wherever possible. For simplicity and exposition, we express holding as a distinct feature.

$$\boxed{T/T \Rightarrow T} \quad \text{Type merging}$$

$$\frac{
 \begin{array}{l}
 P = P_1 \wedge P_2 \\
 C = C_1 \wedge C_2 \\
 \hline
 P_1 \ C_1/P_2 \ C_2 \Rightarrow P \ C
 \end{array}
 \quad
 (\text{hold} \Rightarrow)
 \frac{
 \begin{array}{l}
 T_1 \Rightarrow T_2/T_3 \quad T_2 \downarrow /T'_3 \Rightarrow T'_1 \\
 \Delta, x : T_3 \vdash e \Rightarrow T \dashv \Delta', x : T'_3
 \end{array}
 }{
 \Delta, x : T_1 \vdash \text{hold}[x : T_2](e) \Rightarrow T \dashv \Delta', x : T'_1
 }
 }{
 }$$

The typing rule for hold depends on a notion of *type merging* $T/T \Rightarrow T$, which captures how two separate permissions to an object may be combined. Type merging is defined in terms of the \wedge and \wedge relations, where \wedge is the analogue of \wedge for subpermissions. For example, if we know that variable x has both type $\text{full}(C) \ C$ and $\text{pure}(\text{Object}) \ C$, then we can merge those types, and safely conclude that x can be typed at $\text{full}(C) \ C$.

For space reasons, the rest of the technical development of hold (e.g. translation and dynamic semantics) is omitted, but the details can be found in the technical report [28].

4 Gradual Featherweight Typestate

The previous section presents the essence of GFT: an FJ-like calculus where classes model states, with an update operation to dynamically change the state of objects; types encode states as well as permissions and state guarantees, and the type system is both flow-sensitive and deterministic. Even if features like hold increase the expressiveness of the type system, there are still cases where it is necessary to resort to dynamic assertions about typestates. This section presents the support for such assertions, as well as gradual typing.

Type assertions GFT supports an assert expression for explicitly changing the type assumptions of a variable.

$$e ::= \dots \mid \text{assert}\langle T \rangle(x)$$

Its type synthesis rule is as follows.

$$(\text{assert} \Rightarrow) \frac{}{\Delta, x : T_1 \vdash \text{assert}\langle T_2 \rangle(x) \Rightarrow \text{Void} \dashv \Delta, x : T_2}$$

Assert is like a cast, but instead of returning a value of the given type it changes the type of the target variable⁸. When $T_1 <: T_2$, the assert is statically safe; otherwise, a runtime check is required (see Section 6).

⁸ In fact, assertions are strictly more powerful than casts: casts can be implemented using assertions.

$$\begin{array}{c}
 \boxed{T \Rightarrow T/T} \quad \text{Type Splitting} \qquad \boxed{T \lesssim T} \quad \text{Consistent Subtyping} \\
 \hline
 T \Rightarrow \text{Dyn}/T \qquad \frac{T_1 <: T_2}{T_1 \lesssim T_2} \qquad \frac{T \neq \text{Dyn}}{\text{Dyn} \lesssim T}
 \end{array}$$

Fig. 5. Hybrid Permission Management Relations

$$\begin{array}{c}
 (\text{ctx}_d \Leftarrow) \frac{T \neq \text{Dyn}}{\Delta, x : \text{Dyn} \vdash x \Leftarrow T \dashv \Delta, x : \text{Dyn}} \\
 (\text{update}_d \Rightarrow) \frac{\text{fields}(C_2) = \overline{T_2} \overline{f} \quad \Delta \vdash x_2 \Leftarrow \overline{T_2} \dashv \Delta', x_1 : \text{Dyn}}{\Delta \vdash x_1 \Leftarrow C_2(\overline{x_2}) \Rightarrow \text{Void} \dashv \Delta' \downarrow, x_1 : \text{Dyn}}
 \end{array}$$

Fig. 6. Select Dynamic Typing Rules

Gradual typing To support gradual typing, GFT provides a type for dynamically checked values.

$$T ::= \dots \mid \text{Dyn}$$

The type system treats the Dyn type with greater leniency: type checks on Dyn objects are deferred to runtime.

To account for these dynamic features, Figure 5 presents several adjustments to the type system from the last section. First, type splitting is extended to account for Dyn. In particular, any reference can split off a Dyn without affecting its original type or permissions. Type consistency is extended analogously.

Following Siek and Taha [24], consistent subtyping $T \lesssim T$ extends subtyping to support using Dyn-typed values wherever another type is expected and vice-versa. Consistent subtyping explicitly states that $\text{Dyn} \lesssim T$, but also $T \lesssim \text{Dyn}$ because type splitting now forces $T <: \text{Dyn}$. In accordance, the $(\hat{e} \Leftarrow)$ type checking rule from the last section now uses \lesssim in place of $<:$.

Figure 6 presents some extra typing rules that are needed to account for uses of Dyn-typed references. The $(\text{ctx}_d \Leftarrow)$ rule says that a Dyn-typed variable can be checked at any type. Note that $x : \text{Dyn}$ can be synthesized and checked at Dyn by the $(\text{ctx} \Rightarrow)$ and $(\text{ctx} \Leftarrow)$ rules respectively. The $(\text{update}_d \Rightarrow)$ rule accounts for updating a dynamically typed variable. The type system checks that the arguments to the constructor are suitable, but the checks on the target of the update are deferred to runtime (see Section 6).

5 Internal Language

GFT’s semantics are defined by type-directed translation to GFTIL, an internal language that makes the details of dynamic permission management explicit.

5.1 Syntax

GFTIL is structured much like GFT but elaborates several concepts (Figure 7). First, the internal language introduces explicitly dynamic variants e_d of some

$o \in \text{OBJECTREFS}$	
$l \in \text{INDIRECTREFS}$	
$s ::= x \mid l$	(simple exprs)
$b ::= x \mid l \mid o$	(bare expr)
$e ::= e_s \mid e_d$	(expressions)
$e_s ::= b \mid \text{void} \mid s[T \Rightarrow T/T] \mid \text{new } C(\bar{s})$	(static exprs)
$\quad \mid \text{let } x = e \text{ in } e \mid \text{release}[T](s) \mid s.f \mid s.m(\bar{s})$	
$\quad \mid s.f :=: s \mid s \leftarrow C(\bar{s}) \mid \text{assert}\langle T \gg T \rangle(s)$	
$e_d ::= s.d.f \mid s.d.m(\bar{s}) \mid s.f :=:_d s$	(dynamic exprs)
$\quad \mid s \leftarrow_d C(\bar{s}) \mid \text{assert}_d\langle T \gg T \rangle(s)$	
$\Delta ::= \overline{b : T}$	(type context)

Fig. 7. Internal Language Syntax

operations from the source language. Static variants are ensured to be safe by the type system; dynamic variants require runtime checks. Second, many expressions in the language carry explicit type information. This information is used to dynamically account for the flow of permissions as the program runs. These type annotations play a role in both the type system and the dynamic semantics.

Finally, GFTIL adds several constructs that only occur at runtime. Object references and indirect references point to runtime objects⁹. GFTIL is also in A-normal form, though at runtime the arguments to expressions are generalized to simple expressions s : variable names or indirect references. Reference expressions come in two forms. A bare reference b signifies a variable or reference that is never used again. In contrast, a splitting reference $s[T \Rightarrow T/T]$ explicitly specifies the starting type, result type, and the residual type of the reference. The $\text{release}[T](s)$ expression explicitly releases a reference and its permissions, after which it can no longer be used.

5.2 Static Semantics

Because of GFTIL's explicit form, its typing judgment $\Delta \vdash e : T \dashv \Delta$ does not need to be bidirectional. Furthermore, its typing rules use the same permission and type management relations as the source language. GFTIL's typing rules explicitly and strictly encode permission flow by checking the input context Δ to force their arguments s to have *exactly* the type required. GFTIL's dynamic semantics uses this encoding to track permissions.

Figure 8 presents some of GFTIL's typing rules. For space reasons, we only present the rules for invoke, update and assert, together with their dynamically-typed variants. The (invoke) rule matches a method's arguments exactly against the method signature. Each argument's output type is dictated by the method's output states. The (update) rule almost mirrors GFT's update rule except that its argument types must exactly match the class field specifications. The (assert)

⁹ Object references correspond to heap pointers; indirect references are an artifact that facilitates the type safety proof (see Section 5.4).

$$\begin{array}{c}
\text{(invoke)} \frac{mdecl(m, C_1) = T_r \ m(\overline{T_2 \gg T'_2})[P_1 \ C_1 \ \gg \ T'_1]}{\Delta, s_1 : P_1 \ C_1, s_2 : \overline{T_2} \vdash s_1.m(\overline{s_2}) : T_r \ \dashv \ \Delta \downarrow, s_1 : T'_1, s_2 : \overline{T'_2}} \\
\text{(invoke}_d\text{)} \frac{}{\Delta, s_1 : \text{Dyn}, s_2 : \overline{\text{Dyn}} \vdash s_1.d(\overline{s_2}) : \text{Dyn} \ \dashv \ \Delta \downarrow, s_1 : \text{Dyn}, s_2 : \overline{\text{Dyn}}} \\
\text{(update)} \frac{k \in \{ \text{full}, \text{shared} \} \quad C_2 <: D \quad fields(C_2) = \overline{T \ f}}{\Delta, s_1 : k(D) \ C_1, s_2 : \overline{T} \vdash s_1 \leftarrow C_2(\overline{s_2}) : \text{Void} \ \dashv \ \Delta \downarrow, s_1 : k(D) \ C_2} \\
\text{(update}_d\text{)} \frac{fields(C_2) = \overline{T \ f}}{\Delta, s_1 : \text{Dyn}, s_2 : \overline{T} \vdash s_1 \leftarrow_d C_2(\overline{s_2}) : \text{Void} \ \dashv \ \Delta \downarrow, s_1 : \text{Dyn}} \\
\text{(assert)} \frac{T_1 \cong T_2}{\Delta, s : T_1 \vdash \text{assert}\langle T_1 \gg T_2 \rangle(s) : \text{Void} \ \dashv \ \Delta, s : T_2} \\
\text{(assert}_d\text{)} \frac{T_1 \cong T_2}{\Delta, s : T_1 \vdash \text{assert}_d\langle T_1 \gg T_2 \rangle(s) : \text{Void} \ \dashv \ \Delta, s : T_2}
\end{array}$$

Fig. 8. Select Internal Language Typing Rules

rule is the safe subset of GFT's rule, though GFTIL's `assert` is explicitly annotated with its argument's source type. The dynamic variants of these expressions enforce very little statically: the `(updated)` rule only checks that the arguments match the constructor, and the `(assertd)` rule applies when the destination type cannot be split from the source type.

5.3 Dynamic Semantics

GFTIL's dynamic semantics, presented in Figure 9, requires several additional syntactic notions, defined below:

$$\begin{array}{l}
C(\overline{o}) \ \overline{P} \in \text{OBJECTS} \\
v ::= \text{void} \mid o \quad (\text{values}) \\
\mu \in \text{OBJECTREFS} \rightarrow \text{OBJECTS} \quad (\text{stores}) \\
\rho \in \text{INDIRECTREFS} \rightarrow \text{VALUES} \quad (\text{environments}) \\
\mathbb{E} ::= \square \mid \text{let } x = \mathbb{E} \text{ in } e \quad (\text{evaluation contexts})
\end{array}$$

Expressions in the language evaluate to values, including `void` and object references `o`. Stores μ associate object references to objects. The novelty of GFTIL is that an object in the store $C(\overline{o})$ is annotated with the set of outstanding permissions for references to that object, \overline{P} .

In addition to the store, the dynamic semantics uses a second heap, which we call the *environment* ρ , that mediates between variable references and the object store. In the source language, two variables could refer to the same object in the store, but each can have different permissions to that object. The environment tracks these differences at runtime. It maps indirect references l

$\mu, \rho, e \rightarrow \mu, \rho, e$

 Dynamic Semantics

$$\begin{array}{c}
 \text{(invoke)} \frac{\mu(\rho(l_1)) = C(\overline{o}) \overline{P} \quad \text{method}(m, C) = T_r \ m(\overline{T}_x \gg \overline{T}'_x \ x) \ [T_t \gg T'_t] \ \{ \text{return } e; \}}{\mu, \rho, l_1.m(\overline{l}_2) \rightarrow \mu, \rho, [l_1, \overline{l}_2/\text{this}, \overline{x}]} \\
 \\
 \text{(invoke}_d) \frac{\mu(\rho(l_1)) = C(\overline{o}) \overline{P} \quad \text{mdecl}(m, C) = T_r \ m(\overline{T}_x \gg \overline{T}'_x) \ [T_t \gg T'_t] \quad |\overline{T}_x| = |\overline{l}_2|}{\mu, \rho, l_1.d.m(\overline{l}_2) \rightarrow \mu, \rho, \text{assert}_d\langle \text{Dyn} \gg T_t \rangle(l_1); \text{assert}_d\langle \text{Dyn} \gg T_x \rangle(l_2); \\ \text{let } \text{ret} = l_1.m(\overline{l}_2) \text{ in } \text{assert}\langle T'_t \gg \text{Dyn} \rangle(l_1); \\ \text{assert}\langle T'_x \gg \text{Dyn} \rangle(l_2); \text{assert}\langle T_r \gg \text{Dyn} \rangle(\text{ret}); \\ \text{ret}} \\
 \\
 \text{(update)} \frac{\mu(\rho(l_1)) = C(\overline{o}) \overline{P} \quad \text{fields}(C) = \overline{T} \ \overline{f} \quad \mu' = \mu[\rho(l_1) \mapsto C'(\overline{\rho}(l_2)) \overline{P}] \quad - \overline{o} : \overline{T}}{\mu, \rho, l_1 \leftarrow C'(\overline{l}_2) \rightarrow \mu', \rho, \text{void}} \\
 \\
 \text{(update}_d) \frac{\mu(\rho(l_1)) = C(\overline{o}) \overline{P} \quad D_g = \bigwedge \{ D \mid k(D) \in \overline{P} \} \quad C' <: D_g}{\mu, \rho, l_1 \leftarrow_d C'(\overline{l}_2) \rightarrow \mu, \rho, \text{assert}_d\langle \text{Dyn} \gg \text{shared}(D_g) \ C \rangle(l_1); \\ l_1 \leftarrow C'(\overline{l}_2); \\ \text{assert}\langle \text{shared}(D_g) \ C' \gg \text{Dyn} \rangle(l_1)} \\
 \\
 \text{(assert)} \frac{\mu' = \mu - \rho(l) : T + \rho(l) : T'}{\mu, \rho, \text{assert}\langle T \gg T' \rangle(l) \rightarrow \mu', \rho, \text{void}} \quad \text{(assert}_d \text{v)} \frac{\rho(l) = \text{void}}{\mu, \rho, \text{assert}_d\langle \text{Dyn} \gg \text{Void} \rangle(l) \rightarrow \mu, \rho, \text{void}} \\
 \\
 \text{(assert}_d \text{o)} \frac{\rho(l) = o \quad \mu' = \mu - o : T + o : P' \ C' \quad \mu'(o) = C(\overline{o}) \overline{P} \quad C <: C' \quad \overline{P} \text{ compatible}}{\mu, \rho, \text{assert}_d\langle T \gg P' \ C' \rangle(l) \rightarrow \mu', \rho, \text{void}}
 \end{array}$$

Fig. 9. Select Internal Language Dynamic Semantics Rules

to values v . The dynamic semantics of GFTIL is defined as transitions between store/environment/expression triples¹⁰.

Figure 9 presents some select dynamic semantics rules of GFTIL. Certain rules use two helper functions for tracking permissions in the heap, whose definitions are straightforward and as such omitted for brevity. Permission addition $+$ augments the permission set for a particular object in the heap. Conversely, permission subtraction $-$ removes a permission from the set of tracked permissions for an object. The (invoke) rule is standard. The (update) rule looks up the object references for the target reference and the arguments to the class constructor, replaces the store object for the target reference with the newly constructed object, and releases the permissions held by the fields of the old object. The (assert) rule uses permission addition and subtraction to track permissions, and returns void. Rules for dynamic operators, like (invoke_d) and (update_d), dynamically assert the necessary permissions (using assert_d), defer to the corresponding static operation, and then release the acquired permission (using assert). Finally, the (assert_d) rule confirms dynamically that its type assertion is safe.

¹⁰ The environment serves a purely formal purpose: it supports the proof of type safety by keeping precise track of the outstanding permissions associated with different references to objects at runtime, and is not needed in a practical implementation.

Helper Functions

$$\begin{aligned}
objTypes(\mu, \Delta, \rho, o) &= [T \mid o : T \in types(\mu, \Delta, \rho), T \neq \text{Dyn}] \\
types(\mu, \Delta, \rho) &= fieldTypes(\mu) ++ envTypes(\Delta, \rho) ++ ctxTypes(\Delta) \\
fieldTypes(\mu) &= [o_i : T_i \mid \mu(o'_i) = C(\bar{o}) \bar{P}, fields(C) = \overline{T f}] \\
&\quad o'_i \in dom(\mu) \\
envTypes(\Delta, \rho) &= [o : T \mid \rho(l) = o, l : T \in \Delta] \\
ctxTypes(\Delta) &= [o : T \mid o : T \in \Delta]
\end{aligned}$$

$\mu, \Delta, \rho \vdash o \mathbf{ok}$	$\mu, \Delta, \rho \mathbf{ok}$
Reference Consistency	Global Consistency
$ \begin{array}{l} \mu(o) = C(\bar{o}) \bar{P} \quad \bar{o}' = fields(C) \\ objTypes(\mu, \Delta, \rho, o) = \overline{k(E) D} \\ C <: \bar{D} \quad \frac{k(E) \text{ compatible}}{k(E) = P} \\ \hline \mu, \Delta, \rho \vdash o \mathbf{ok} \end{array} $	$ \begin{array}{l} ran(\rho) \subset dom(\mu) \cup \{\text{void}\} \\ dom(\Delta) \subset dom(\rho) \cup dom(\mu) \\ \{l \mid (l : \text{Void}) \in \Delta\} \subset \{l \mid \rho(l) = \text{void}\} \\ \{l \mid (l : k(D) C) \in \Delta\} \subset \{l \mid \rho(l) = o\} \\ \mu, \Delta, \rho \vdash dom(\mu) \mathbf{ok} \\ \hline \mu, \Delta, \rho \mathbf{ok} \end{array} $

Fig. 10. Permission-Consistency Relations

5.4 Type safety

GFTIL’s type safety proof must account for the outstanding permissions for each object o and verify that they are mutually compatible. Figure 10 presents the definitions used for this. The *fieldTypes*, *ctxTypes*, and *envTypes* functions accumulate outstanding type information for objects in the store from the fields of objects, the type context, and the environment respectively. The *objTypes* function selects just the permission-carrying types for a particular object reference o . These definitions use square brackets to express list comprehensions, and $++$ to express list concatenation.

The *objTypes* function is used to define *reference consistency*, the judgment that an object in the store and all references to it are sensible. A consistent object reference points to an object that has the proper number of fields, and all references to it are well-formed, mutually compatible, and tracked in the store.

Reference consistency is used in turn to define *global consistency*, which establishes the mutual compatibility of a store-environment-context triple. Global consistency implies that every object reference in the store satisfies reference consistency, that every reference in the type context is accounted for in the store and environment, and that **Void** and object-typed indirect references ultimately point to **void** values and object references respectively¹¹. Note that global consistency and permission tracking take into account even objects that are no longer reachable in the program. To recover permissions, a program must explicitly release the fields of an object before it becomes unreachable.

These concepts contribute to the statement (and proof) of type safety.

Theorem 1 (Progress). *If e is a closed expression, $\mu, \Delta, \rho \mathbf{ok}$, and $\Delta \vdash e : T \rightarrow \Delta'$, then only one of the following holds:*

¹¹ Dyn references may point to either **Void** or object references.

$$\begin{array}{c}
 \text{(invoke}\Rightarrow\text{)} \frac{\text{mdecl}(m, C_1) = T \text{ m}(\overline{T_x \gg T'_x})[T_t \gg T'_t] \quad \text{coerce}(x_1, P_1 \ C_1 \lesssim T_t) = e_1^T \quad \text{coerce}(x_2, T_2 \lesssim T_x) = e_2^T}{\Delta, x_1 : P_1 \ C_1, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) \Rightarrow T \rightsquigarrow \overline{e_1^T; e_2^T}; x_1.m(\overline{x_2}) \dashv \Delta \downarrow, x_1 : T'_t, x_2 : \overline{T'_x}} \\
 \\
 \text{(invoke}_d \Rightarrow\text{)} \frac{\text{coerce}(x_2, T_2 \lesssim \text{Dyn}) = e_2^T}{\Delta, x_1 : \text{Dyn}, \overline{x_2 : T_2} \vdash x_1.m(\overline{x_2}) \Rightarrow \text{Dyn} \rightsquigarrow \overline{e_2^T}; x_1.d.m(\overline{x_2}) \dashv \Delta \downarrow, x_1 : \text{Dyn}, \overline{x_2 : \text{Dyn}}} \\
 \\
 (\hat{e} \Leftarrow) \frac{\Delta \vdash \hat{e} \Rightarrow T_1 \rightsquigarrow e_1^T \dashv \Delta' \quad \text{coerce}(\text{ret}, T_1 \lesssim T_2) = e_2^T}{\Delta \vdash \hat{e} \Leftarrow T_2 \rightsquigarrow \text{let ret} = e_1^T \text{ in } e_2^T; \text{ret} \dashv \Delta'} \\
 \\
 \text{(update}\Rightarrow\text{)} \frac{\text{fields}(C_2) = \overline{T_2 f} \quad \Delta \vdash x_2 \Leftarrow T_2 \rightsquigarrow e_2^T \dashv \Delta', x_1 : k(D) \ C \quad k \in \{\text{full, shared}\} \quad C_2 <: D}{\Delta \vdash x_1 \leftarrow C_2(\overline{x_2}) \Rightarrow \text{Void} \rightsquigarrow \text{let } x'_2 = e_2^T \text{ in } x_1 \leftarrow C_2(x'_2) \dashv \Delta' \downarrow, x_1 : k(D) \ C_2} \\
 \\
 \text{(update}_d \Rightarrow\text{)} \frac{\text{fields}(C_2) = \overline{T_2 f} \quad \Delta \vdash x_2 \Leftarrow T_2 \rightsquigarrow e_2^T \dashv \Delta', x_1 : \text{Dyn}}{\Delta \vdash x_1 \leftarrow C_2(\overline{x_2}) \Rightarrow \text{Void} \rightsquigarrow \text{let } x'_2 = e_2^T \text{ in } x_1 \leftarrow_d C_2(x'_2) \dashv \Delta' \downarrow, x_1 : \text{Dyn}} \\
 \\
 \text{(assert}\Rightarrow\text{)} \frac{T \Rightarrow T'}{\Delta, x : T \vdash \text{assert}\langle T' \rangle(x) \Rightarrow \text{Void} \rightsquigarrow \text{assert}\langle T \gg T' \rangle(x) \dashv \Delta, x : T'} \\
 \\
 \text{(assert}_d \Rightarrow\text{)} \frac{T \not\Rightarrow T'}{\Delta, x : T \vdash \text{assert}\langle T' \rangle(x) \Rightarrow \text{Void} \rightsquigarrow \text{assert}_d\langle T \gg T' \rangle(x) \dashv \Delta, x : T'}
 \end{array}$$

Fig. 11. Select Translation Rules from GFT to GFTIL

- e is a value;
- $\mu, \rho, e \rightarrow \mu', \rho', e'$ for some μ', ρ', e' ;
- $e = \mathbb{E}[e_d]$ and μ, ρ, e is stuck.

The last case of the progress theorem holds when a program is stuck on a failed dynamically checked expression. All statically checked expressions make progress.

Theorem 2 (Preservation). *If $\Delta \vdash e : T \dashv \Delta'$, and μ, Δ, ρ **ok**, and $\mu, \rho, e \rightarrow \mu', \rho', e'$, then $\Delta'' \vdash e' : T \dashv \Delta'$ and μ', Δ'', ρ' **ok** for some Δ'' .*

6 Source to Target Translation

The dynamic semantics of GFT are defined by augmenting its type system to generate GFTIL expressions. The type checking and synthesis judgments become $\Delta \vdash e_1 \Leftarrow T \rightsquigarrow e_2^T \dashv \Delta'$ and $\Delta \vdash e_1 \Rightarrow T \rightsquigarrow e_2^T \dashv \Delta'$ respectively, where e_1 is a GFT expression and e_2^T is its corresponding GFTIL expression. Figure 11 presents some of these rules. We use the T superscript to disambiguate GFTIL expressions as needed. Several rules use the *coerce* partial function, which translates consistent subtyping assertions $T \lesssim T$ into variable assertions:

$$\begin{array}{l}
 \text{coerce}(x, T_1 \lesssim T_2) = \text{assert}\langle T_1 \gg T_2 \rangle(x) \quad \text{if } T_1 <: T_2 \\
 \text{coerce}(x, \text{Dyn} \lesssim T) = \text{assert}_d\langle \text{Dyn} \gg T \rangle(x) \quad \text{if } T \neq \text{Dyn}
 \end{array}$$

Most of the translations are straightforward, and follow similar patterns. For instance, the $(\text{update} \Rightarrow)$ rule, which applies when the target of the update is statically typed, let-binds all of the arguments to the object constructor so as to extract the exact permissions that it needs before calling GFTIL’s static update. The $(\text{update}_d \Rightarrow)$ rule, in contrast, applies when the target of the update is dynamically typed. It translates to a dynamic update operation \leftarrow_d , but is otherwise the same. Operations on dynamically typed objects translate to dynamic operations. Other rules like $(\text{assert} \Rightarrow)$ simply use the typing rule to ascertain the intended type annotations for the corresponding GFTIL expression.

As intended, the translation rules preserve well-typing:

Theorem 3 (Translation Soundness).

If $\Delta \vdash e \Leftrightarrow T \rightsquigarrow e^T \dashv \Delta'$ then $\Delta \vdash e^T : T \dashv \Delta'$.

This theorem extends straightforwardly to whole programs.

7 Discussion

Related Work. A lot of research has been done on tpestates since they were first introduced by Strom and Yemini [26]. Most tpestate analyses are whole-program analyses, which makes them very flexible in handling aliasing. Approaches based on abstract interpretation (e.g. [12]) rely on a global alias analysis and generally assume that the protocol implementation is correct and only verify client conformance. Naeem and Lhoták [20] developed an analysis for checking tpestate properties over multiple interacting objects. These global analyses typically run on the complete code base, only once a system is fully implemented, and are time consuming.

Fugue [10] was the first modular tpestate verification system for object-oriented software. It tracks objects as “not aliased” or “maybe aliased”; only “not aliased” objects can change state. Bierhoff and Aldrich [4] extended this approach by supporting more expressive method specifications based on linear logic [16]. They introduce the notion of access permissions in order to allow state changes even in the presence of aliasing. They also use fractions, first proposed by Boyland [8], to support patterns like borrowing and adoption [9]. The Plural tool supports modular tpestate checking with access permissions for Java. It has been used in a number of practical studies [5]. Although Plural introduced state guarantees, this paper provides their first formalization.

Recent work on distributed session types [15] provides essentially the same expressiveness as Plural, but with protocols expressed in the structural setting of a process algebra instead of the setting of nominal tpestates. It considers communication over distributed channels as well as object protocols, but does not allow aliasing for objects with protocols.

The above approaches do not address tpestate-oriented programming, as they are not integrating tpestates within the programming model, but rather overlay static tpestate analysis on top of an existing language. TSOP has been recently proposed by Aldrich et al. [2]; its defining characteristic is supporting

run-time changes to the representation of objects in the dynamic semantics and type system. The programming language Plaid¹² is the first language to integrate typestates in the core programming model. Saini et al. [23] recently developed the first core calculus for a TSOP language; their language is object-based and relies on structural types. Gradual Featherweight Typestate builds on this work but adapts it to a class-based, nominal approach with shared permissions and state guarantees for reasoning about typestate in the presence of aliasing. Earlier work related to TSOP includes the Fickle system [11], which can change the class of an object at runtime, but has limited ability to reason about the states of an object’s fields.

This work also builds upon existing techniques for partial typing, like hybrid typing [18] and gradual typing [25, 24, 6]. Gradual Featherweight Typestate is a considerable advance in this sense, by showing how to gradually check flow-sensitive resources in a modular fashion. Recently, Bodden [7] presented a hybrid approach to typestate checking. A static typestate analysis is performed to avoid unnecessary instrumentation of programs for monitoring typestates at runtime. While the hybrid perspective is shared with this work, the proposed analysis is global.

Ahmed et al. [1] define a core functional programming language that supports *strong updates*, i.e. changing the type of an object in a reference cell. Similarly to our approach, it uses linear typing. They present two languages, L3, and extended L3. L3 allows aliasing, but only has exclusive access, through a capability: only one reference can read/write to an object. In contrast, full, shared and pure permissions allow for more varied aliasing patterns. Extended L3 allows recovering a capability, but the programmer must provide a proof that no other capabilities exist to the reference cell.

Future Work. Gradual Featherweight Typestate is at the core of the Plaid language design project at CMU. We are integrating other access permissions from Bierhoff and Aldrich [4]. Most importantly, we are exploring ways to extend the power of the static type system in order to avoid resorting to dynamic asserts. An example of such an extension is permission borrowing [9], which, if specified in method signatures, avoids having to dynamically reassert permissions after “lending” them to a sub-computation. The language we present here already includes one such refinement, namely `hold`, used to hold some permissions to a reference while a sub-computation is performed. Importantly, it remains an outstanding research question if the cost of dynamic permission checking can be amortized over the number of permission checks. As it now stands, enabling dynamic permission checking mandates a fully-instrumented runtime semantics to keep track of permissions. In Plaid, we intend to address this with reference counting. Standard optimization techniques like deferred increments [3] and update coalescing [19] will be applied. We believe these techniques will reduce reference count overhead to a small percentage of runtime, as it does for garbage collection, and will study this empirically in future. The formalism presented

¹² Under development at CMU: <http://plaid-lang.org>

here establishes a baseline from which to explore this capability and develop new models for permission tracking.

Conclusion. Gradual Featherweight Typestate (GFT) is a nominal core calculus for typestate-oriented programming. By introducing typestate directly into the language and extending its type system with support for gradual typing, state abstractions can be implemented directly, stronger program properties can be enforced statically, and when necessary dynamic checks can be introduced seamlessly. In particular GFT supports a rich set of access permissions together with state guarantees for substantial reasoning about typestate in the presence of aliasing. This work paves the way for further gradual approaches by showing how to modularly and gradually check flow-sensitive resources.

Acknowledgments. We thank the members of the Plaid research group at Carnegie Mellon University, especially Darpan Saini and Nels Beckman, as well as the anonymous reviewers, for feedback on this work.

References

1. Ahmed, A., Fluet, M., Morrisett, G.: L3: A linear language with locations. *Fundamenta Informaticae* 77(4), 397–449 (2007)
2. Aldrich, J., Sunshine, J., Saini, D., Sparks, Z.: Typestate-oriented programming. In: *Proc. Onward! 2009*, pp. 1015–1022. ACM, New York (2009)
3. Baker, H.G.: Minimizing reference count updating with deferred and anchored pointers for functional data structures. *SIGPLAN Not.* 29, 38–43 (1994)
4. Bierhoff, K., Aldrich, J.: Modular typestate checking of aliased objects. In: *Proc. Conference on Object-oriented Programming Systems and Applications*, pp. 301–320. ACM, New York (2007)
5. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical API protocol checking with access permissions. In: Drossopoulou, S. (ed.) *ECOOP 2009*. LNCS, vol. 5653, pp. 195–219. Springer, Heidelberg (2009)
6. Bierman, G., Meijer, E., Torgersen, M.: Adding dynamic types to c^\sharp . In: D’Hondt, T. (ed.) *ECOOP 2010*. LNCS, vol. 6183, pp. 76–100. Springer, Heidelberg (2010)
7. Bodden, E.: Efficient hybrid typestate analysis by determining continuation-equivalent states. In: *Proc. International Conference on Software Engineering*, pp. 5–14. ACM, New York (2010)
8. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) *SAS 2003*. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
9. Boyland, J., Retert, W.: Connecting effects and uniqueness with adoption. In: *Symposium on Principles of Programming Languages*, pp. 283–295. ACM, New York (2005)
10. DeLine, R., Fähndrich, M.: Typestates for objects. In: Vetta, A. (ed.) *ECOOP 2004*. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)

11. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: Fickle: Dynamic object re-classification. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, Springer, Heidelberg (2001)
12. Fink, S.J., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.* 17(2), 1–34 (2008)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Professional Computing Series. Addison-Wesley, Reading (1994)
14. Garcia, R., Wolff, R., Tanter, É., Aldrich, J.: Featherweight Typestate. Technical Report CMU-ISR-10-115, Carnegie Mellon University (July 2010)
15. Gay, S., Vasconcelos, V., Ravara, A., Gesbert, N., Caldeira, A.: Modular session types for distributed object-oriented programming. In: Symposium on Principles of programming languages, pp. 299–312. ACM, New York (2010)
16. Girard, J.-Y.: Linear logic. *Theor. Comput. Sci.* 50(1), 1–102 (1987)
17. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23(3), 396–450 (2001)
18. Knowles, K., Flanagan, C.: Hybrid type checking. *ACM Trans. Program. Lang. Syst.* 32(2), 6:1–6:34 (2010)
19. Levanoni, Y., Petrank, E.: An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.* 28, 1–69 (2006)
20. Naeem, N.A., Lhoták, O.: Typestate-like analysis of multiple interacting objects. In: Proc. Conference on Object-oriented programming systems languages and applications, pp. 347–366. ACM, New York (2008)
21. Pierce, B.C., Turner, D.N.: Local type inference. *ACM Trans. Program. Lang. Syst.* 22(1), 1–44 (2000)
22. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. *Lisp Symb. Comput.* 6(3-4), 289–360 (1993)
23. Saini, D., Sunshine, J., Aldrich, J.: A theory of typestate-oriented programming. In: Formal Techniques for Java-like Programs (2010)
24. Siek, J.G., Taha, W.: Gradual typing for objects. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007)
25. Siek, J., Taha, W.: Gradual typing for functional languages. In: Proc. Scheme and Functional Programming Workshop (September 2006)
26. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.* 12(1), 157–171 (1986)
27. Walker, D.: Substructural type systems. In: Pierce, B. (ed.) *Advanced Topics in Types and Programming Languages*, ch. 1, pp. 3–43. MIT Press, Cambridge (2005)
28. Wolff, R., Garcia, R., Tanter, É., Aldrich, J.: Gradual Featherweight Typestate. Technical Report CMU-ISR-10-116R, Carnegie Mellon University (July 2010)

Maintaining Database Integrity with Refinement Types

Ioannis G. Baltopoulos¹, Johannes Borgström², and Andrew D. Gordon²

¹ University of Cambridge

² Microsoft Research

Abstract. Taking advantage of recent advances in automated theorem proving, we present a new method for determining whether database transactions preserve integrity constraints. We consider check constraints and referential-integrity constraints—extracted from SQL table declarations—and application-level invariants expressed as formulas of first-order logic. Our motivation is to use static analysis of database transactions at development time, to catch bugs early, or during deployment, to allow only integrity-preserving stored procedures to be accepted. We work in the setting of a functional multi-tier language, where functional code is compiled to SQL that queries and updates a relational database. We use refinement types to track constraints on data and the underlying database. Our analysis uses a refinement-type checker, which relies on recent highly efficient SMT algorithms to check proof obligations. Our method is based on a list-processing semantics for an SQL fragment within the functional language, and is illustrated by a series of examples.

1 Introduction

This paper makes a case for the idea that database integrity should be maintained by static verification of transactional code, rather than by relying on checks at run time. We describe an implementation of this idea for relational databases, where schemas are defined using SQL table descriptions, and updates are written in a functional query language compiled to SQL. Our method relies on a semantics of SQL tables (including constraints) using refinement types, and a semantics of SQL queries in terms of list processing. We describe a series of database schemas, the implementation of transactions in the .NET language F#, and the successful verification of these transactions using the refinement-type checker Stateful F7. Like several recent tools, Stateful F7 relies in part on pushing verification conditions to external SMT solvers, provers whose effectiveness has recently improved at a remarkable rate. Our aim is to initiate the application of modern verification tools for functional languages to the problem of statically-verified database transactions, and to provide some evidence that the idea is at last becoming practical.

1.1 Background: Database Integrity Constraints

SQL table descriptions may include various sorts of constraints, as well as structural information such as base types for columns.

A *check constraint* is an assertion concerning the data within each row of a table, expressed as a Boolean expression.

A *primary key constraint* requires that a particular subset, the *primary key*, of the columns in each row of the table identifies the row uniquely within the table. A key consisting of multiple column labels is called a *composite key*. A *uniqueness constraint* is similar to a primary key constraint but based on a single column.

A *foreign key constraint* requires that a particular subset, a *foreign key*, of the columns in each row of the table refers uniquely to a row in the same or another table. Satisfaction of primary key and foreign key constraints is known as *referential integrity*.

To illustrate these constraints by example consider a table recording marriages between persons, represented by integer IDs. A key idea is that the marriage of A and B is encoded by including both the tuples (A, B) and (B, A) in the table.

An Example Table with Integrity Constraints: Marriage

```
create table [Marriage](
  [Spouse1] [int] not null unique,
  [Spouse2] [int] not null,
  constraint [PK_Marriage] primary key ([Spouse1],[Spouse2]),
  constraint [FK_Marriage] foreign key ([Spouse2], [Spouse1])
    references [Marriage] ([Spouse1], [Spouse2]),
  constraint [CK_Marriage] check (not([Spouse1] = [Spouse2])))
```

The two columns `Spouse1` and `Spouse2` in the `Marriage` table store non-null integers. Database integrity in this example amounts to three constraints: marriage is monogamous (you cannot be in two marriages), symmetric (if you marry someone they are married to you), and irreflexive (you cannot marry yourself).

- The uniqueness constraint on the column `Spouse1` asserts that nobody is `Spouse1` in two different marriages, hence enforcing monogamy.
- The primary key constraint `PK_Marriage` in conjunction with the self-referential foreign key constraint `FK_Marriage` asserts that whenever row (A, B) exists in the table, so does the row (B, A) , hence enforcing symmetry.
- The check constraint `CK_Marriage` asks that nobody is married to themselves, hence enforcing irreflexivity.

A buggy transaction on this table may violate its constraints. The sorts of bugs we aim to detect include the following: (1) insertion of null in `Spouse1` or `Spouse2` (violating the **not null** type annotation); (2) inserting (A, C) when (A, B) already exists (violating the uniqueness constraint); (3) inserting (A, B) but omitting to insert (B, A) (violating the foreign key constraint); and (4) inserting (A, A) (violating the check constraint). We aim to eliminate such integrity violations by static analysis.

1.2 Background: Multi-tier Functional Programming

We consider the common situation where database updates are not written directly in SQL, but instead are generated from a separate programming language via some object-relational mapping. In particular, we consider database transactions expressed in the functional language F# [28], but compiled to SQL for efficient execution in the relational backend. This is an instance of *multi-tier functional programming*, where a single functional program is split across tiers including the web server and the database.

Our mapping is based on three ideas:

- (1) We model SQL table definitions as F# types: the whole database is a record type `db` consisting of named tables, where each table is a list of records, corresponding to the rows of the table.
- (2) We provide the user with *standard* functions for create, read, update, and delete operations on each table. We also allow user-supplied *custom* SQL stored procedures, and provide F# functions to call these procedures. Both standard and custom functions are implemented as SQL queries, and can be thought of as imperative actions on a global state of type `db`.
- (3) Users write a transaction as an F# function that interacts with the database by calling a sequence of standard SQL functions and custom stored procedures.

To illustrate point (1), we model our example table definition with the following F# types, where the whole database `db` is a record with a single field holding the marriages table, which itself is a list of rows.

```
type marriage_row = { m_Spouse1:int; m_Spouse2:int; }
type db = { marriages: marriage_row list; }
```

A row (A, B) is represented by the record:

```
{ m_Spouse1=A; m_Spouse2=B; }
```

The marriage of A and B is represented by the list:

```
[{ m_Spouse1=A; m_Spouse2=B }; { m_Spouse1=B; m_Spouse2=A }]
```

Regarding point (2), we have (among others) the following standard queries as F# functions:

- `hasKeyMarriage` (A, B) computes whether a row with primary key (A, B) exists in the marriages table.
- `deleteMarriage` (A, B) deletes the row with primary key (A, B) from the marriages table, if it exists.

We have no user-supplied custom SQL queries for the marriages example, but show such queries in some of our later examples.

Actual transactions (point (3) above) are written as functional code. The following example of a user-written transaction is to dissolve a marriage. Given two spouses A and B , we have to check whether the rows (A, B) and (B, A) exist in the database and remove them both.

An Example Transaction: Divorce

```
let divorce_ref (A,B) =
  if hasKeyMarriage(A, B) then
    deleteMarriage(A, B);
    deleteMarriage(B, A);
    Some(true)
  else Some(false)
```

The body of the function is an expression returning a value of type `bool option`.

- `Some true` means there was a marriage successfully removed, and we commit;
- `Some false` means there was no marriage to remove, and we commit;
- `None` would mean that the transaction failed and any updates are to be rolled back (a return value not illustrated by this code).

The code above takes care to check that a marriage between A and B already exists before attempting to delete it, and also to remove both (A, B) and (B, A) . Instead, careless code might remove (A, B) but not (B, A) . Assuming that the foreign key constraint on the marriage table is checked dynamically, such code would lead to an unexpected failure of the transaction. If dynamic checks are not enabled (for instance since the underlying database engine does not support deferred consistency checking) running invalid code would lead to data corruption, perhaps for a considerable duration. Our aim is to detect such failures statically, by verifying the user written code with a refinement-type checker.

1.3 Databases and Refinement Types

The values of a *refinement type* $x:T\{C\}$ are the values x of type T such that the formula C holds. (Since the formula C may contain values, refinement types are a particular form of dependent type.) A range of refinement-type checkers has recently been developed for functional languages, including DML [31], SAGE [14], F7 [4], DSolve [24], Fine [27], and Dminor [6], most of which depend on SMT solvers [23].

A central idea in this paper is that refinement types can represent database integrity constraints, and SQL table constraints, in particular. For example, the following types represent our marriage table.

SQL Table Definitions as Refinement Types:

```

type marriage_row = { m_Spouse1:int; m_Spouse2:int }
type marriage_row_ref = m:marriage_row {CK_Marriage(m)}
type marriage_table_ref = marriages:marriage_row_ref list
  { PKtable_Marriage(marriages) ^ Unique_Marriage_Spouse1(marriages) }
type State = { marriage:marriage_table_ref }
type State_ref = s:State {FK_Constraints(s)}

```

The refinement types use predicate symbols explained informally below. We give formal details later on.

- `CK_Marriage(m)` means the record m satisfies the check constraint `[CK_Marriage]`.
- `PKtable_Marriage(marriages)` means the list of records `marriages` satisfies the primary key constraint with label `[PK_Marriage]`.
- `Unique_Marriage_Spouse1(marriages)` means `marriages` satisfies the uniqueness constraint on column `[Spouse1]`.
- `FK_Constraints(s)` means the database s satisfies the foreign key constraint with label `[FK_Marriage]`.

1.4 Transactions and the Refined State Monad

The state monad is a programming idiom for embedding imperative actions within functional programs [29]. Pure functions of type `State` $\rightarrow T * \text{State}$ represent computations that interact with a global state; they map an input state to a result paired with an output state. The *refined state monad* [13][20][8] $[(s_0)C_0]x:T [(s_1)C_1]$ is the enrichment of the state monad with refinement types as follows:

$$[(s_0)C_0]x:T [(s_1)C_1] \triangleq s_0:\text{State}\{C_0\} \rightarrow x:T * s_1:\text{State}\{C_1\}$$

The formula C_0 is a precondition on input state s_0 , while the formula C_1 is a postcondition on the result x and output state s_1 .

A new idea in the paper is to represent SQL queries and transactions as computations in a refined state monad, with the refinement type `State` being a record with a field for each table in the database, as above. For example, the function `divorce_ref` has the following type, where the result of the function is a computation in the refined state monad.

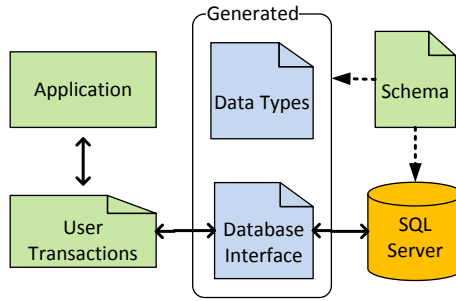
```
val divorce_ref: (int × int) →
  {(s) FK_Constraints(s)} r:bool option {(t) r ≠ None ⇒ FK_Constraints(t)}
```

The return type states that if the function is called in a state s satisfying the foreign key constraints, and it terminates, then it returns a value r of type `bool option`. Moreover, if $r \neq \text{None}$, then the state t after the computation terminates satisfies the foreign key constraints. The type reflects that the code performs sufficient dynamic checks that it never causes a dynamic failure, and that it returns `None` whenever it leaves the database in an inconsistent state. The case of the function returning `None` is caught by a transaction wrapper (not shown here) which then aborts the transaction, rolling back the database to its initial state.

By checking with refinement types, we aim to catch buggy code that terminates with `Some b`, intending to commit and return b , but that does not in fact re-establish the database invariants. In particular, we would catch code that removes say (A, B) but not (B, A) , as it does not re-establish the foreign key constraint `FK_Constraints(t)`.

1.5 An Architecture for Verified Database Transactions

We verify a series of example user transactions, according to the diagram below. Each example starts from a database schema in SQL. From the schema our tool generates refinement types to model the database, and also a functional programming interface for a set of pre-packaged stored procedures in SQL (for actions such as querying and deleting items by key, exemplified by the functions `lookupMarriage` etc mentioned above). Against this interface, the user writes transactional code (exemplified by the function `divorce_ref` mentioned above) in F#, which is invoked from their application. We verify the code of the user transactions using the typechecker `Stateful F7` [8], which implements the refined state monad on top of the typechecker `F7` [4]. Additionally, not shown in the diagram, in some examples the schema may also include queries written directly as custom SQL stored procedures; we can also verify these queries by mapping SQL into F#.



Our examples are as follows.

- (1) Marriages (see above and Section 3).

We have a single table `Marriage(Spouse1, Spouse2)` with integrity expressed using SQL table constraints. We describe verifiable transactions to create and delete marriages.

- (2) Order processing (see Section 4).

We have tables `Orders(OrderID, CustomerID, Name, Address)` and `Details(OrderID, ProductID, UnitPrice, Quantity)` with integrity expressed using primary key, foreign key, and check constraints. We show that an `addOrder` function, which creates an order with a single detail row, respects all these constraints.

- (3) Heap data structure (see Section 5).

We have a table `Heap(HeapID, Parent, Content)` where each row represents a node in a heap data structure. SQL constraints do not suffice to specify the integrity invariant of this data structure, so we add user-defined constraints written in first-order logic. We verify that integrity is preserved by recursive functions to push and pop elements, which make use of user-defined stored procedures `getRoot` and `getMinChild`.

1.6 Contributions of the Paper

Our main contribution is to interpret SQL table descriptions as refinement types, and database updates as functional programs in the refined state monad, so as to verify, by refinement-type checking, that updates preserve database integrity. Hence, verification of the F# and SQL source code proceeds by sending a series of verification conditions in first-order logic to an automatic theorem prover.

Our source code is in the .NET language F#, but our method could be recast for other functional multi-tier languages, such as Links [11], HOP [25], or FLAPJAX [17], and also for object-oriented programming models such as LINQ [19]. We use the type-checker Stateful F7, but we expect our approach to queries and transactions would easily adapt to related verifiers for functional code with state such as Why [13] or YNOT [20], and indeed to verifiers for imperative code, such as those using Boogie [2].

The idea of static verification of database transactions goes back to the 1970s, to work on computing the weakest precondition needed for a transaction to succeed [16,9,26,5]. Theorem proving technology has improved considerably since the idea of static verification of database transactions was first mooted, and an implication

of our work is that the idea is at last becoming practical. Moreover, the success of languages with functional features such as F#, Scala [21], and indeed recent versions of C# with closures, is compelling evidence for the significance of functional programming as an object-oriented technology. Hence, our work lays a foundation for statically verifiable database access from mainstream object-oriented platforms.

A technical report includes additional details [1].

2 A Tool to Model SQL with Refinement Types

This section fleshes out the architecture diagram of our system.

Section 2.1 describes the details of the SQL schemas input by our system, including both the data definition part defining the structure of tables, and also the data manipulation part of queries invoked from stored procedures.

Section 2.2 details how our tool generates data types and a database interface from a schema. The database interface consists of a set of F# functions with types, including preconditions and postconditions, specified in the syntax of Stateful F7. When generating the database interface, our tool automatically includes functions to access a set of standard queries, as well as functions to access any custom stored procedures included in the schema.

Section 2.3 gives a symbolic reference implementation for the generated database interface. The symbolic implementation relies on list processing in a similar fashion to Peyton Jones and Wadler [22] and serves as a formal semantics for the interface. We trust, but do not formally verify, that the behaviour of the symbolic implementation corresponds to its interface, as well as to our concrete implementation which works by sending queries to an actual SQL database.

Finally, Section 2.4 extends our schema syntax with the ability to write integrity constraints directly as first-order predicates.

2.1 SQL Schemas: Tables and Stored Procedures

Let c range over constants, x over variables and f, g over table column names. Then, boolean expressions and value expressions occurring within SQL queries are defined by the syntax below. Value expressions include (boolean, integer, and string) constants, binary operations, variables and table field names. Boolean expressions include equations between value expressions, comparisons, conjunction, disjunction, and negation.

Values and Expressions

B	$::= E = E \mid E \odot E \mid B \vee B \mid B \wedge B \mid \neg B$	Boolean expression
\odot	$::= < \mid <= \mid > \mid >=$	Comparison operator
E	$::= E \oplus E \mid c \mid x \mid f$	Value expression
\oplus	$::= + \mid - \mid * \mid /$	Binary operator

A table declaration defines a table t , and gives a name f_i and type T_i to each of its columns. To indicate uniqueness constraints, each column has a tag u_i , either **unique** or empty. SQL supports several data types; in this work, we only consider **Boolean**, **Int** and

String data types which are defined as their counterparts in F#. We may interpret more complex types using appropriate encodings and refinements of the three basic types. We assume that each table has exactly one primary key, which may be composite, exactly one check constraint, and no multiple foreign key references to the same table. (SQL syntax allows multiple check constraints, but these may be conjoined to produce a single check constraint).

Data Definition

$DT ::=$	Table declaration
table $t (u_i f_i : T_i)^{i \in 1..n},$	name and fields
primary key $\bar{g},$	primary key
check $B,$	check constraint
$\kappa_1, \dots, \kappa_m$	foreign keys
$\kappa ::=$ foreign key \bar{f} references $t(\bar{g})$	
$T ::=$ Boolean Int String	Type

The syntax of supported SQL queries includes those necessary for selecting, inserting and deleting rows from a table. Let t denote an SQL table name and let \bar{f} be a shorthand for f_1, \dots, f_n (all the columns of the table) and \bar{g} be a shorthand for g_1, \dots, g_m (denoting some of the f_i).

Data Manipulation

$Q ::= QS \mid QI \mid QD \mid QU$	Query
$QS ::=$	Select query
select [top 1] \bar{g}	selector
from t	source
where B	criterion
[order by f { asc desc }]	ordering
$QI ::=$	Insert query
insert into t	target table
(f_1, \dots, f_n)	table fields
values (E_1, \dots, E_n)	field values
$QD ::=$	Delete query
delete from t	target table
where B	criterion
$QU ::=$	Update query
update t set	target table
$(g_1, \dots, g_m) =$	table fields
(E_1, \dots, E_m)	field values
where B	criterion

A QS query filters all the rows of a table t , based on a boolean criterion B , and projects the selected fields \bar{g} . The result of a select query is a table of rows matching the criterion. We consider only **select** queries that contain **top 1** if and only if they contain an **order by** clause. In this case, the resulting table is ordered in either ascending or descending order based on the single field f , and the first element of the table is returned

as the result. A *QI* query adds a row consisting of the values E_1, \dots, E_n in table t . In **insert into** $t (f_1, \dots, f_n)$ **values** (E_1, \dots, E_n) , we expect each E_i to be either a variable or a constant. The result of a *QI* query is a number indicating the count of successful insertions. A *QD* query removes from table t all the rows matching the boolean criterion B . Again, the result is a number indicating the count of successful deletions. Finally, a *QU* query modifies fields \bar{g} to contain values \bar{E} for all rows of table t that match the boolean condition B .

The SQL schema syntax includes constructs for databases, tables, procedures, and constraints. A schema is a named tuple of declarations. A declaration can be either a procedure or a table. A procedure abstracts a query Q by giving a name h and making it depend on parameters a_1, \dots, a_n . We assume that in a procedure declaration, the query Q only contains variables from \bar{a} .

SQL Schema

$S ::=$ schema $s(DT_i^{i \in 1..n}, DP_j^{j \in 1..m})$	Schema
$DP ::=$ procedure $h(a_i : T_i)^{i \in 1..n} Q$	Procedure declaration

In subsequent sections, we adopt a convenient syntax for advanced queries and assume standard encodings of these syntactic forms in terms of the core query syntax. For example, multi-row insertion is defined in terms of multiple single-row insertions and the star (*) syntax in *QS* queries corresponds to explicitly naming all the columns in the table, in order of their appearance in the table declaration.

2.2 Generating Types and Database Interfaces from Schemas

Our tool maps an SQL schema S to a Stateful F7 module $\llbracket S \rrbracket$ by assembling a series of type definitions, predicate definitions, and function signatures. This section describes each of the components in turn.

Translation from S in SQL to $\llbracket S \rrbracket$ in Stateful F7:

Let $\llbracket S \rrbracket$ be the Stateful F7 module obtained from schema S by concatenating the type and function definitions displayed below: (S1) types from schema declarations; (S2) refinement formulas from constraints; (S3) signatures of standard functions; (S4) signatures of custom functions.

First, in (S1) we fix a type for table declarations and the global type `State` of the refined state monad used by Stateful F7. Second, in (S2) we define logical axioms which correspond to the database constraints. Third, we give types to queries and procedures that manipulate the global state. As discussed earlier, expressions get *computation types* of the form $[(s_0)C_0]x:T[(s_1)C_1]$. Finally, in (S3) and (S4) we give signatures of standard API functions and custom procedures for manipulating the global state. We use **val** $f : T$ to give a type to a function in the API.

Below, we assume a fixed schema s defined by S . For every table t in s , assume the definition **table** $t (u_i f_i : T_i)^{i \in 1..n}$, **primary key** \bar{g} , **check** B , $\kappa_1, \dots, \kappa_l$. Given table t , the

translation algorithm works as follows. We generate the type t_key as a tuple of the corresponding types of the primary key fields and let T_f , the type of field f , be given by T_i when $f = f_i$. For each row in the table we create an unrefined record type t_row with the labels corresponding to the column names from the table definition. To associate a **check** constraint with each table row, we refine the row type with the formula $CK_t(row)$ (defined below) and create the refinement type t_row_ref . Values of this type represent rows in the table t for which the check constraint holds. The table t itself is modelled as a list of refined rows (t_table). Finally, we refine the table type by associating the primary key constraint formula $PK_table_t(tab)$ (defined below in (S2)) with it. Values of this type represent tables for which the primary key constraint holds. Basic types translate directly to their equivalents in Stateful F7. We deal with **not null** and **null** constraints by declaring nullable types as option types.

We can now proceed to the definition of the type corresponding to the database (for a single schema). Without loss of generality, assume that the tables t_1, \dots, t_n belong to the fixed schema s . The database type is a record of refined tables. In the refined state type, the refinement asserts that values of this type will satisfy the foreign key constraints on the database. The normal state type does not have this refinement, denoting that top-level constraints may temporarily be invalidated. A valid transaction may assume that the foreign key constraints hold, and must enforce them on exit, but may internally temporarily violate the constraints.

(S1) Types from Table Declarations

```

type t_key = Tg1 × ... × Tgm
type t_row = {f1:T1; ...; fn:Tn}
type t_row_ref = row:t_row {CK_t(row)}
type t_table_ref = tab:t_row_ref list {PK_table_t(tab) ∧ ∧_{u_i=unique} Unique_t.fi(tab)}
type State = { (t_i : t_i_table_ref) }^{i ∈ 1..n}
    
```

We now define logical predicates corresponding to SQL table constraints. We assume a translation $L[\cdot]_r$ from SQL boolean and value expressions to logical formulas and terms; the translation is homomorphic except for the base case $L[f]_r \triangleq r.f$.

- $CK_t(row)$ means the check constraint of table t holds of the tuple row .
- $PK_t(r,k)$ means the primary key of row r of table t is k .
- $PK_table_t(tab)$ means the contents tab of table t satisfies its primary key constraint.
- $Unique_t.f(tab)$ means the contents tab of table t satisfies the uniqueness constraint for field f .
- $FK_t_u(tab1,tab2)$ means the contents $tab1$ of table t satisfies the foreign key constraint with reference to the contents $tab2$ of table u .
- $FK_Constraints(db)$ means all foreign key constraints in the database db are satisfied.

In the table below, the Stateful F7 keyword **assume** introduces a universally quantified axiom to define each new predicate symbol.

(S2) Refinement Formulas from Constraints

assume $\forall \text{row}. \text{CK_t}(\text{row}) \Leftrightarrow \text{L}[B]_{\text{row}}$
assume $\forall \text{row}, \bar{x}. \text{PK_t}(\text{row}, (\bar{x})) \Leftrightarrow \bigwedge_i x_i = \text{row}.g_i$
assume $\forall \text{tab}. \text{PKtable_t}(\text{tab}) \Leftrightarrow \forall \text{row1}, \text{row2}.$
 $(\text{Mem}(\text{row1}, \text{tab}) \wedge \text{Mem}(\text{row2}, \text{tab}) \wedge \text{PK_t}(\text{row1}, (\text{row2}.g_1, \dots, \text{row2}.g_m))) \Rightarrow \text{row1} = \text{row2}$
assume $\forall \text{tab}. \text{Unique_t_f}(\text{tab}) \Leftrightarrow \forall \text{row1}, \text{row2}.$
 $(\text{Mem}(\text{row1}, \text{tab}) \wedge \text{Mem}(\text{row2}, \text{tab}) \wedge \text{row1}.f = \text{row2}.f) \Rightarrow \text{row1} = \text{row2}$
assume $\forall \text{tab1}, \text{tab2}. \text{FK_t_u}(\text{tab1}, \text{tab2}) \Leftrightarrow \forall x. \text{Mem}(x, \text{tab1}) \Rightarrow (\exists y. \text{Mem}(y, \text{tab2}) \wedge$
 $\bigwedge_i x.f_i = y.g_i)$
 if $\exists \kappa_i =$ **foreign key** $f_1 \dots f_m$ **references** $u(g_1 \dots g_m)$
assume $\forall s. \text{FK_Constraints}(s) \Leftrightarrow \bigwedge_{t, u} \text{FK_t_u}(s.t, s.u)$

(A limitation of our semantics for uniqueness constraints and primary keys is that they allow duplicate copies of the same row; the limitation could be lifted by more elaborate semantics but we leave this for future work).

Now that we have translated SQL data declarations, we may proceed to the query and data manipulation languages. The variable s in the translation represents the entire database record, and therefore the expression $s.t$ projects the table t over which the query is performed. A simple select query does not modify the state, and returns a list whose elements are exactly the rows in the table matching the select condition. We write $T_{\bar{g}}$ for the tuple $T_1 \times \dots \times T_n$ where T_i is the type of field g_i and $n = |\bar{g}|$. A select **top 1** query also does not modify the state, and returns a list which is either empty or contains one element from the table that matches the select criterion and is less than (or greater than, not shown) any other such element. An insert query may only be called if inserting the row does not invalidate any table constraints, and the new table after running the query is the old table with the inserted row prepended to it. A delete query modifies the corresponding table to contain only those rows not matching the query condition. An update query also modifies the table to contain exactly those rows that do not match the where clause, or the updated version of the rows that do.

Types of SQL Queries

$\mathcal{T}[\text{select } \bar{g} \text{ from } t \text{ where } B] \triangleq$
 $[(s) \text{ l}: T_{\bar{g}} \text{ list } [(s') s=s' \wedge (\forall x. \text{Mem}(x, t) \Leftrightarrow \exists r. \text{L}[B]_r \wedge \bar{r}.g = \bar{x}.g \wedge \text{Mem}(r, s.t))]]$
 $\mathcal{T}[\text{select top 1 } \bar{g} \text{ from } t \text{ where } B \text{ order by } f \text{ asc}] \triangleq$
 $[(s) \text{ l}: T_{\bar{g}} \text{ list } [(s') s=s' \wedge ((l = [] \wedge (\forall r. \text{Mem}(r, s.t) \Rightarrow \neg \text{L}[B]_r)) \vee$
 $(\exists x. \text{L}[B]_x \wedge \text{Mem}(x, s.t) \wedge l = [\{\bar{g} = \bar{x}.g\}] \wedge (\forall r. \text{L}[B]_r \wedge \text{Mem}(r, s.t) \Rightarrow r.f >= x.f)))]]$
 $\mathcal{T}[\text{insert into } t(f_1, \dots, f_n) \text{ values } (E_1, \dots, E_n)] \triangleq$
 $[(s) \text{ PKtable_t}(\{\bar{f} = \text{L}[E]_r\} :: (s.t))] \text{ unit } [(s') s' = \{s \text{ with } t = \{\bar{f} = \bar{E}\} :: (s.t)\}]$
 $\mathcal{T}[\text{delete from } t \text{ where } B] \triangleq$
 $[(s) \text{ int } [(s') \exists t'. s' = \{s \text{ with } t = t'\} \wedge (\forall r. \text{Mem}(r, t') \Leftrightarrow \text{L}[\neg B]_r \wedge \text{Mem}(r, s.t))]]$
 $\mathcal{T}[\text{update t set } \bar{g} = \bar{E} \text{ where } B] \triangleq$
 $[(s) \text{ int } [(s') \exists t'. s' = \{s' \text{ with } t = t'\} \wedge (\forall x. \text{Mem}(x, t') \Leftrightarrow (\text{Mem}(x, s.t) \wedge \neg \text{L}[B]_r \vee$
 $(\exists r. \text{Mem}(r, s.t) \wedge \text{L}[B]_r \wedge x = \{r \text{ with } \bar{g} = \text{L}[\bar{E}]_r\}))]]$

The standard API defines functions that look up the existence of keys inside a table, generate fresh keys for a table, checks that an unrefined row satisfies the constraints, inserts a refined row in a table, deletes a row from a table, and updates a row in a table. The function `fresh.t` is only defined when the primary key is non-composite, that is, the constraint contains a single field f , and indeed is an integer. The `lookup.t` function takes a numeric key and returns true if it exists inside the table; the `fresh.t` function generates a new key that does not exist inside the table; the `check.t` function takes an unrefined row and makes sure that all the check constraints are satisfied for it; the `insert.t` function takes a refined row to be inserted in a table and starting from a state that satisfies all the primary key and foreign key constraints performs the insertion; the `delete.t` function takes a key for a table and removes the associated row from the table.

(S3) Signatures of Standard Functions

```

PKfresh_t(tab, (x̄)) ≜ ∀r. Mem(r, tab) ⇒ ∨i(xi ≠ r.gi)
PKexists_t(tab, (x̄)) ≜ ∃r. Mem(r, tab) ∧ ∧i(xi = r.gi)

val hasKey_t: k:t_key →
  [(s) True] b:bool [(s') s=s' ∧ (b=false ⇒ PKfresh_t(s.t,k)) ∧ (b=true ⇒ PKexists_t(s.t,k))]
val lookup_t: k:t_key →
  [(s) True] o:t_row_ref option [(s') s=s' ∧ (o=None ⇒ PKfresh_t(s.t,k)) ∧
    (∀r. o=Some(r) ⇒ Mem(r,s.t) ∧ PK_t(r,k))]
val fresh_t: unit → [(s) True] k:t_key [(s') s=s' ∧ PKfresh_t(s.t,k)]
val check_t: r:t_row → [(s) True] b:bool [(s') s=s' ∧ (b=true ⇒ CK_t(r))]
val insert_t: r:t_row_ref → [(s) True] b:bool [(s') (b=false ⇒ s=s') ∧
  (b=true ⇒ s' = {s with t = r:(s.t) })]
val update_t: r:t_row_ref → [(s) True] b:bool [(s') (b=false ⇒ s=s') ∧
  (b=true ⇒ ∃t'. s' = {s with t = t'} ∧
    (∀x. Mem(x,t') ⇔ (Mem(x,s.t) ∧ ¬x.ḡ = r.ḡ) ∨ (x=r ∧ ∃y. Mem(y,s.t) ∧ y.ḡ = r.ḡ))]
val delete_t: k:t_key → [(s) True] unit [(s') (b=false ⇒ s=s') ∧
  (b=true ⇒ ∃t'. s' = {s with t = t'} ∧ (∀x. Mem(x,t') ⇔ (Mem(x,s.t) ∧ ¬x.ḡ = r.ḡ))]
    
```

To complete the four parts of the definition of $\llbracket S \rrbracket$, each custom stored procedure explicitly listed in the schema S is translated to a function signature as follows.

(S4) Signatures of Custom Functions

```

[[procedure h (ai : Ti)i∈1..n Q]] ≜
  val h : a1 : ℱ[T1] → ... → an : ℱ[Tn] → ℱ[Q]
    
```

2.3 Reference Implementation of Database Interface

The dynamic semantics for the subset of SQL that we consider follows Peyton Jones and Wadler [22]. In the following, we assume standard `map` and `filter` functions on lists, and also functions `max` and `min` that select the maximum and minimum of a list of

orderable values. As a convention, we use \vec{f} for the full tuple of columns, and \bar{g} for a subset of the columns. The translation $F[\cdot]_r$, from SQL boolean and value expressions to F# expressions is homomorphic except for the base case $F[f]_r \triangleq r.f$.

Semantics of SQL Queries

```

[[select  $\bar{g}$  from  $t$  where  $B$ ]]  $\triangleq$ 
  let  $s = \text{get}()$  in map ( $\text{fun } \vec{f} \rightarrow \bar{g}$ )(filter ( $\text{fun } r \rightarrow F[B]_r$ ) ( $s.t$ ))
[[select top 1  $\bar{g}$  from  $t$  where  $B$  order by  $f$  asc]]  $\triangleq$ 
  match [[select  $f$  from  $t$  where  $B$ ]] with []  $\rightarrow$  [] |  $xs \rightarrow$ 
  let  $m = \max(xs)$  in [hd([[select  $\bar{g}$  from  $t$  where  $(f = m) \wedge B$ ]])]
[[insert into  $t$  ( $f_1, \dots, f_n$ ) values ( $E_1, \dots, E_n$ )]]  $\triangleq$ 
  let  $s = \text{get}()$  in set { $s$  with  $t = \{f_1 = E_1, \dots, f_n = E_n\} :: (s.t)$ }
[[delete from  $t$  where  $B$ ]]  $\triangleq$ 
  let  $s = \text{get}()$  in set { $s$  with  $t = [[select * from t where  $\neg B$ ]]}$ }
[[update  $t$  set  $\bar{g} = \bar{E}$  where  $B$ ]]  $\triangleq$ 
  let  $s = \text{get}()$  in
  let  $t1 = [[select * from t where  $\neg B$ ]]$  in
  let  $tB = [[select * from t where  $B$ ]]$  in
  let  $t2 = \text{map} (\text{fun } r \rightarrow \{r \text{ with } \overline{F[g]_r} = \overline{F[E]_r}\}) tB$  in
  set { $s$  with  $t = t1 @ t2$ }

```

To translate a simple *QS* query, we first obtain the current database and project the table t we are interested in. We then filter every row r using the translation of the boolean condition B . Finally, we map a projection function, which selects the required subset of columns \bar{g} , onto the filtered result. The translation of a *QS* query with **top** and **order by** first narrows the result set using the boolean criterion B . If no rows match the criterion, we simply return the empty list. If multiple rows match the criterion, we find the maximum value of the field f within any row and store that to a temporary variable m . We, finally, use a simple *QS* query to find all the rows that satisfy B for which the field f has the value m and return the head of the list. The translation of a *QI* query involves getting the current database value, and immediately writing it back with the new row being prepended to the existing table. The translation of a *QD* query follows a similar pattern; we get the current database and immediately write back a table consisting of all the rows that do not match the boolean condition. The translation of a *QU* query first saves the initial state, as well as the rows of table t that do not match the criterion B . We then extract the rows that match the criterion B , and map the update over them. Finally, the modified state is written back.

Here is the semantics for the API in F#, with appeal to our semantics of SQL in F#. Below we write $pk(t)$ for the non-empty tuple of field names making up the primary key of table t , and we write $ck(t)$ for the check constraint of table t .

Semantics of API Functions

```

let hasKey_t k = [[select * from t where pk(t)=k]] ≠ []
let lookup_t k =
  match [[select * from t where pk(t)=k]] with
  | [r] → Some r
  | _ → None
let fresh_t () =
  genKey_t [[select top 1 pk(t) from t where true order by pk(t) asc]]
let check_t r = F[[ck(t)]]_r
let insert_t r = let nrows = [[insert into t (f1,...,fn) values (r.f1,...,r.fn)]] in 1 == nrows
let delete_t k = let nrows = [[delete from t where pk(t)=k]] in ()
let update_t k r = (delete_t k; insert_t r)

```

The user code is written in F# and can be executed symbolically against the reference implementation of the database access API above. The same user code is typechecked against the F7 interface and linked against the concrete implementation of the API functions that use a relational database.

2.4 Extension with Application Constraints

We extend the SQL table syntax from Section 2.1 in order to allow user-specified invariants, written in first-order logic. We also replace the definition of refined tables, and the definition of the global database constraint `FK_Constraints` as follows.

User Constraints

```

κ ::= ... | p                                p is a unary predicate symbol
D ::= ... | p                                p is a unary predicate symbol

type t_table_ref = tab:t_table{PKtable_t(tab) ∧ ∧_{i=1...k} p_i^†(tab)}
assume ∀db. FK_Constraints(db) ⇔ ∧_{t,u} FK_t_u(s.t, s.u) ∧ ∧_i p_i^D

```

User constraints $p(x)$, where x will be instantiated either by a table or the entire database, are defined by a user-specified first order logic formula C_p that can contain boolean expressions, quantifiers, and other axiomatized predicates. When defining these formulas, care must be taken to avoid introducing inconsistencies—any program satisfies an inconsistent specification. To help with this, F7 can work in a debug mode which attempts to prove `false` every time an axiom or a refined value is introduced.

3 Completing the Marriages Example

Our goal is to type-check application code that accesses the database and to ensure that it respects the database constraints. To achieve this we need a model of the database,

the tables and the constraints inside the host language of the application. Based on the rules from section 2.2 we translate the `Marriages` table declaration from section 1.1 and generate the appropriate F7 data types with refinements. We now give the complete translation of the marriage example.

3.1 Database Schema

We here repeat the definition of the refined data types corresponding to the marriage table and its rows.

Marriage Data Types

```

type marriage_row = { m_Spouse1:int; m_Spouse2:int }
type marriage_row_ref = m:marriage_row {CK_Marriage(m)}
type marriage_table_ref = marriages:marriage_row_ref list
  { PKtable_Marriage(marriages)  $\wedge$  Unique_Marriage_Spouse1(marriages) }
type State = { marriage:marriage_table_ref }
type State_ref = s:State {FK_Constraints(s)}

```

The check constraint, primary key constraint, uniqueness constraint and foreign key constraint are defined below as first-order logical formulas, using the keyword **assume**. We define two auxiliary predicates: `PK_Marriage(m, k)` states that the primary key of row `m` is `k`, and `FK_Constraints(s)` states that all foreign key constraints (of which there is only one) are satisfied for the database `s`. The predicate `Mem(r,t)` checks if row `r` is present in table `t`.

SQL Constraints as Formulas

```

assume  $\forall x,y. \text{CK\_Marriage}((x, y)) \Leftrightarrow x \neq y$ 
assume  $\forall m,k. \text{PK\_Marriage}(m, k) \Leftrightarrow k = (m.m\_Spouse1, m.m\_Spouse2)$ 
assume  $\forall xs. \text{PKtable\_Marriage}(xs) \Leftrightarrow$ 
   $\forall x,m. \text{Mem}(x, xs) \wedge \text{Mem}(m, xs) \wedge \text{PK\_Marriage}(x, (m.m\_Spouse1, m.m\_Spouse2))$ 
   $\Rightarrow x = m$ 
assume  $\forall l. \text{Unique\_Marriage\_Spouse1}(l) \Leftrightarrow$ 
   $\forall x,y. \text{Mem}(x, l) \wedge \text{Mem}(y, l) \wedge x.m\_Spouse1 = y.m\_Spouse1 \Rightarrow x = y$ 
assume  $\forall d. \text{FK\_Constraints}(d) \Leftrightarrow \text{FK\_Marriages\_Marriages}(d.marriages, d.marriages)$ 
assume  $\forall marriages', marriages. \text{FK\_Marriages\_Marriages}(marriages, marriages') \Leftrightarrow$ 
   $\forall x. \text{Mem}(x, marriages') \Rightarrow$ 
   $\exists u. \text{Mem}(u, marriages) \wedge \text{PK\_Marriage}(u, (x.m\_Spouse2, x.m\_Spouse1))$ 

```

3.2 Access Function API

From the database schema, our tool also generates data manipulation functions which carry preconditions and postconditions corresponding to the database constraints on

their arguments. We generate two implementations of these functions: one that works on the abstract model, and one that works on the actual SQL server database via ADO.Net.

The following code fragment contains the type signatures of the automatically generated functions for the marriage example.

Specification of API Functions

```

val checkMarriage: r:marriage_row → [(s)] b:bool [(s')(s = s' ∧ b = true ⇒ CK_Marriage(r))]
val hasKeyMarriage :
  k:(int × int) → [(s)] b:bool [(s')( s = s' ∧
                                b = false ⇒ PK_Marriages_Fresh(s.marriages, k) ∧
                                b = true ⇒ PK_Marriages_Exists(s.marriages, k))]

val deleteMarriage :
  k:(int × int) → [(s) PK_Marriages_Exists(s.marriages, k)] unit [(s')
  ContainsiffNotPKMarriage(s, s', k)]

val insertMarriage :
  r:marriage_row_ref → [(s)] b:bool [(s')(
  b = true ⇒ s'.marriages = r :: s.marriages ∧
  b = false ⇒ s = s')]

assume (∀s,t,k. (ContainsiffNotPKMarriage(s, t, k) ⇔
  (∀x. (Mem(x, t.marriages) ⇔ (Mem(x, s.marriages) ∧ not PK_Marriage(x, k))))))
assume (∀marriages,spouse1,spouse2. (PK_Marriages_Fresh(marriages, (spouse1, spouse2))
  ⇔ (∀x. (Mem(x, marriages) ⇒ (spouse1 ≠ x.m_Spouse1 ∨ spouse2 ≠ x.m_Spouse2))))))
assume (∀marriages,spouse1,spouse2. (PK_Marriages_Exists(marriages, (spouse1, spouse2))
  ⇔ (∃x. ((Mem(x, marriages) ∧ spouse1 = x.m_Spouse1) ∧ spouse2 = x.m_Spouse2))))

```

3.3 User-Written Transactions

In addition to the divorce transaction seen in section 3.1, the user also writes a transaction to marry two people. Note that the foreign key constraint (symmetry) is temporarily invalidated between the two row insertions. The verification will ensure that it is properly reestablished at the end of the transaction.

Marriage Transaction

```

let marry_ref (A,B) =
  if hasKeyMarriage(A,B) then Some(false)
  else if A=B then Some(false)
  else
    insertMarriage {m_Spouse1=A; m_Spouse2=B};
    insertMarriage {m_Spouse1=B; m_Spouse2=A};
    Some(true)

let marry m = doTransact marry_ref m

```


The final line above defines a transaction `marry` by calling the transaction wrapper `doTransact`, which ensures that transactions that may violate database integrity are rolled back. The marriage transaction, wrapped and unwrapped, and the transaction wrapper, have the following types.

Wrapping Transactions

```

type  $\alpha$  transaction = [(s) FK_Constraints(s)] r:  $\alpha$  [(t) FK_Constraints(t)]
type  $\alpha$  preTransact =
  [(s) FK_Constraints(s)] r:  $\alpha$  option
  [(t) r  $\neq$  None  $\Rightarrow$  FK_Constraints(t)]

val marry_ref: int  $\times$  int  $\rightarrow$  bool preTransact
val doTransact: ( $\alpha \rightarrow \beta$  preTransact)  $\rightarrow$   $\alpha \rightarrow$  ( $\beta$  option) transaction

```

A transaction returning type α is a computation, which if run in a state satisfying the foreign key constraints, if it terminates, returns a value of type α in a state that satisfies the foreign key constraints. Similarly, a pre-transaction returning type α is a computation, which if run in a state satisfying the foreign key constraints, and terminating with a return value of type α option different from `None`, preserves the foreign key constraints. To go from a pre-transaction, e.g., `marry_ref` to a transaction, e.g., `marry`, we use the function `doTransact` which rolls back the pre-transaction if it returns `None`.

We verify that the user code above has the types given above by refinement type checking; in particular, we get that the functions `marry` and `toTransact divorce_ref` preserve database integrity.

4 Example: A Simple E-Commerce Application

In this section, we illustrate our approach in the context of a typical e-commerce web shopping cart. A user can add products to their cart, update the number of products or remove items from their order. Each operation must leave the database in a consistent state satisfying all database constraints. An operation either successfully completes the database transaction, leaving the database in a new state, or it aborts the transaction and rolls back all the intermediate modifications, leaving the database in its original state.

We store the shopping cart state across two database tables. The first one (`Orders`) holds order information like customer name and shipping address, while the second one (`Details`) stores specific details about orders, like the codes of the chosen products, their quantities, and their price. A row in the `Details` table represents a unique product in an order. The column `OrderID` is used to associate each order with multiple detail rows.

The following SQL fragment shows the two tables, with their constraints.

SQL Schema

```

create table [Orders](
  [OrderID] [int] not null,
  [CustomerID] [nchar](8) null,
  [ShipName] [nvarchar](40) null,
  [ShipAddress] [nvarchar](60) null,
  constraint [PK_Orders] primary key ([OrderID])
)
create table [Details](
  [OrderID] [int] not null,
  [ProductID] [int] not null,
  [UnitPrice] [money] not null,
  [Quantity] [smallint] not null,
  constraint [PK_Details] primary key ([OrderID], [ProductID]),
  constraint [FK_Details_Orders] foreign key([OrderID])
  references [Orders] ([OrderID]),
  constraint [CK_Quantity] check ((([Quantity]>(0))),
  constraint [CK_UnitPrice] check ((([UnitPrice]>=(0))))

```

The primary key is the compound key created from the `OrderID` and the `ProductID` fields. To ensure referential integrity, we add the constraint that for every row in the `Details` table, the value of the `OrderID` field must exist in a row of the `Orders` table. For data integrity, we ask that for each row in the table, the `Quantity` is positive and the `UnitPrice` is non-negative.

E-Commerce Data Types (partial)

```

type State = { orders : orders_ref; details : details_ref }
type State_ref=d:State{ FK_Constraints(d) }
assume  $\forall d. \text{FK\_Constraints}(d) \Leftrightarrow \text{FK\_Details\_Orders}(d.\text{details}, d.\text{orders})$ 
assume  $\forall ds, os. \text{FK\_Details\_Orders}(ds, os) \Leftrightarrow$ 
   $\forall x. \text{Mem}(x, ds) \Rightarrow \exists u. \text{Mem}(u, os) \wedge \text{PK\_Orders}(u, x.d\_OrderID)$ 

```

Given the types corresponding to table definitions (omitted), we represent a database as a record whose labels correspond to the table names. The label types are the refined table types `orders_ref` and `details_ref`. A refined state `State_ref`, is a database for which the foreign key constraint between the tables holds. The foreign key predicate definition says that for every row `x` of the details table, there exists a row `u` in the orders table, such that the primary key of `u` is equal to the `d_OrderID` field of `x`.

We verify the user defined pre-transaction `addOrder` below.

E-Commerce Transaction

```

let addOrder_ref ord =
  let (customerID, shipName, shipAddress, productID, unitPrice, quantity) = ord in
  let oid = freshOrders () in
  let orders : orders_row =
    {o_OrderID = oid; o_CustomerID = Some(customerID);
     o_ShipName = Some(shipName); o_ShipAddress = Some(shipAddress)} in
  let details : details_row =
    {d_OrderID = oid; d_ProductID = productID;
     d_UnitPrice = unitPrice; d_Quantity = quantity} in
  let rowChecks = checkDetails details in
  if rowChecks then let r = insertDetails details in
    if r then let r' = insertOrders orders in
      if r' then Some(true)
      else None
    else None
  else None

let addOrder ord = doTransact addOrder_ref ord

```

In `addOrder_ref`, since the detail is inserted before the order row, the database passes through a state in which the foreign key constraint is violated. (This code would fail needlessly in some systems, such as SQL Server.) The function `addOrder` uses the library function `doTransact` to wrap `addOrder_ref` with the necessary transaction handling code; type-checking ensures that the transaction is rolled back when necessary to avoid violation of integrity constraints.

5 Example: A Heap-Ordered Tree

This example shows the use of more advanced features of our system, such as user-defined predicates and custom stored procedures. We use a database table to store a heap-ordered tree, where the child nodes store pointers to their parent but not vice versa. We show how to define and typecheck for adding and removing nodes of the heap.

Heap SQL Specification

```

create table [Heap](
  [HeapID] [int] identity (1,1) not null,
  [Parent] [int] not null,
  [Content] [int] not null,
  constraint
  [PK_Heap] primary key CLUSTERED ([HeapID] asc),
  constraint
  [FK_Heap] foreign key ([Parent]) references [Heap] ([HeapID]),
  /×--- UserConstraint TR_isHeap ×/
  /×--- UserConstraint TR_uniqueRoot ×/

```

We add two named application-level invariants (Section 2.4) to the heap table.

- `TR_isHeap` states that the value stored at every node is greater than that at its parent;
- `TR_uniqueRoot` states that any two root nodes are equal.

The first-order formulas expressing the application-level invariant predicates are defined in terms of an auxiliary predicate `TR_isRoot`. This predicate denotes that a given node is the root of a tree, which is defined as the node being its own parent.

User Constraints

```

assume  $\forall d. \text{TR\_isHeap}(d) \Leftrightarrow (\forall x,y. (\text{Mem}(x, d) \wedge \text{Mem}(y,d) \wedge$ 
     $x.h\_Parent = y.h\_HeapID) \Rightarrow x.h\_Content \geq y.h\_Content)$ 
assume  $\forall d. \text{TR\_uniqueRoot}(d) \Leftrightarrow$ 
     $(\forall x,y. (\text{TR\_isRoot}(x,d) \wedge \text{TR\_isRoot}(y,d)) \Rightarrow x = y)$ 
assume  $\forall x,d. \text{TR\_isRoot}(x,d) \Leftrightarrow \text{Mem}(x, d) \wedge x.h\_Parent = x.h\_HeapID$ 

```

We also define two stored procedures: `getRoot` returns a root node of the tree, while `getMinChild` returns the smallest child of a given node.

Custom Stored Procedures

```

create procedure getRoot as
  select top 1 * from Heap
  where HeapID = Parent order by Content asc

create procedure getMinChild @rootID [int] as
  select top 1 * from Heap
  where (Parent = @rootID and HeapID  $\neq$  @rootID)
  order by Content asc

```

The form of these stored procedures is very similar, so we detail the translation of only `getRoot`. Its postcondition is defined as follows. The function can return two different values: the empty list or a list containing one element. If the function returns the empty list, we learn that there is no root element. If one element was returned, the predicate `GetRootResult` states that it satisfies the where clause, and is from the table, and the predicate `GetRootIsMin` states that the returned element is the one with the least value of the elements in the table satisfying the where clause.

getRoot

```

val getRoot : unit  $\rightarrow$  [(s) l:heap_row list
  [(s') s = s'  $\wedge$  GetRootResult(l,s)  $\wedge$ 
  ((l = []  $\wedge$  GetRootNotFound(s))  $\vee$  ( $\exists x. l = [x]$ ))]
assume  $\forall s,x. \text{GetRootNotFound}(s) \wedge \text{Mem}(x,s.\text{heaps}) \Rightarrow$ 
  not (x.h_Parent = x.h_HeapID)
assume  $\forall s,l,x. (\text{GetRootResult}(l,s) \wedge (l = [x])) \Rightarrow$ 
  (x.h_HeapID = x.h_Parent)  $\wedge$  Mem(x, s.heaps)  $\wedge$ 
  PK_Heaps_Exists(s.heaps,x.h_HeapID)  $\wedge$  GetRootIsMin(x,s)
assume  $\forall x,s,r. (\text{GetRootIsMin}(x,s) \wedge r.h\_HeapID = r.h\_Parent$ 
   $\wedge \text{Mem}(r, s.\text{heaps})) \Rightarrow x.h\_Content \geq r.h\_Content$ 

```

In this setting, we define two operations. We can insert a node into the tree, using the function `pushAt_int`, which adds a node with a given value as a child to the nearest ancestor of a given node that has a value less than the value to insert. With `pop_int` we can pop the smallest node off the table, causing its smallest child to bubble up the tree, recursively. This recursive procedure is called `rebalanceHeap`.

Specifications of User Functions

```

val pushAt_int: int×int → bool preTransact
val pop_int: unit → int preTransact
val rebalanceHeap: i:int →
  [(s) FK_Constraints(s) ∧ PK_Heaps_Exists(s.heaps,i) ]
  unit [(t) FK_Constraints(t) ]

```

To push an element, we compare it to the root. If it is smaller, it becomes the new root value, otherwise we store it as a child of the root.

Pushing an Element Onto the Heap

```

let rec pushAt_int (i,v) =
  let node = lookupHeap i in
  let newID = freshHeap () in
  match node with
  | None → None
  | (Some(nodeRow)) →
    let {h_Content=c ; h_HeapID=id ; h_Parent=par} = nodeRow in
    if v > c then
      let r = {h_Content = v ; h_HeapID = newID ; h_Parent = id} in
      if insertHeap r then Some(true) else None
    else
      if hasKeyHeap id then
        if hasKeyHeap par then
          if id = par then
            let nodeRow' = {h_Content=v ; h_HeapID=id ; h_Parent=par} in
            if updateHeap id nodeRow' then
              let r = {h_Content=c ; h_HeapID=newID ; h_Parent=id} in
              if insertHeap r then Some(true) else None
            else None
          else pushAt_int (id,v)
        else None
      else None

```

When popping the root, we use `rebalanceHeap` to let a chain of minimal children “bubble up” one step.

Popping the Root of the Heap

```

let rec rebalanceHeap id =
  let minM = getMinChild(id) in match minM with
  | [] → let res = deleteHeap id in res
  | [minRow] → match minRow with
  | {h_Content=mc; h_HeapID=mid; h_Parent=mpar} →
    if hasKeyHeap mid then
      let r = lookupHeap id in match r with
      | None → ()
      | (Some(u)) → match u with
      | {h_Content=rc ; h_HeapID=rid; h_Parent=rpar} →
        let v = {h_Content = mc; h_HeapID = id ; h_Parent = rpar} in
        updateHeap id v;
        let res = rebalanceHeap mid in res
    else ()

let pop_int () =
  let root = getRoot() in match root with
  | [] → None
  | [rootRow] → match rootRow with
  | {h_Content = c; h_HeapID = id; h_Parent = par} →
    (rebalanceHeap id; Some(c))

```

To verify this more complex example, we needed to add three axioms to the context of the SMT solver. The first axiom states that when updating a row, without changing its primary key, then the same primary keys are present in the database table as before. The second axiom states that if the foreign key constraints hold, and the primary and foreign key fields are unchanged by a single-row update, then the foreign key constraints are not violated. The third axiom states that if a row has no children, then it can be deleted without violating the foreign key constraint.

Axioms

```

assume ∀h,h',k,v,x. UpdateHeap(h',h,k,v) ∧ PK_Heaps_Exists(h,x) ⇒ PK_Heaps_Exists(h',x)
assume ∀h1,h2,x,y. FK_Heaps_Heaps(h1,h1) ∧ Replace(h2,h1,x,y) ∧ x.h_Parent = y.h_Parent
  ∧ x.h_HeapID = y.h_HeapID ⇒ FK_Heaps_Heaps(h2,h2)
assume ∀s,k,s'. FK_Constraints(s) ∧ GetMinChildNotFound(k,s) ∧ DeletedHeap(s,s',k) ⇒
  FK_Constraints(s')

```

Given these axioms, we verify that transactions that add values to or pop values from the tree do not violate the database integrity, including the application-level constraints.

6 Software Architecture and Evaluation

Our implementation consists of a compiler from an SQL schema to a Stateful F7 database interface implementing the translation in Section 2.2, and from an SQL schema to a

Table 1. Lines of user supplied and generated code, and verification information

	User supplied		Generated		Verification	
	transactions	schema	data types	db interface	queries	time
Marriages	38	10	38	48	20	20.890s
E-Commerce	41	23	54	74	16	9.183s
Heap	111	30	54	85	76	80.385s

symbolic implementation of the database in F# implementing the dynamic semantics of Section 2.3. We use the Stateful F7 typechecker to verify the user supplied transactions against the generated interface. Additionally we provide a concrete implementation of the database interface against SQL Server. The core compiler (without Stateful F7) consists of about 3500 lines of F# code split between the SQL parser, the translation rules, and the implementation of the database interface.

We evaluate our approach experimentally by verifying all the examples of this paper; Table 1 summarizes our results. For each example it gives: a) the total number of lines of user supplied code (including the F# transaction code and user-defined predicates, and the SQL schema declaration), b) the number of lines of the automatically generated data types and database interface, and c) the verification information consisting of the number of proof obligations passed to Z3 and the actual verification time. Constraints that affect individual rows or tables like **check**, and **primary key** constraints, unsurprisingly add little time to the verification process. This explains the small verification time of the E-Commerce example, despite having more tables and **check** constraints than the other examples. On the other hand uniqueness, foreign key constraints, and arbitrary user constraints require disproportionately more time to verify.

We express constraints in first-order logic with a theory of uninterpreted function symbols and linear arithmetic. The main challenge when working with first-order solver like Z3 is quantifier instantiation. In certain examples like heap, we found that Z3 was unable to prove the automatically generated predicates. As a result and to assist Z3 with its proof obligations, our compiler implements some additional intermediate predicates. In particular, for universally quantified formulas, we gave predicate names to quantified subformulas such that superfluous instantiations might be avoided. For existentially quantified formulas, Z3 sometimes has problems constructing the appropriate witness, and we instead were forced to add an axiom that referred to predicates that abstracted quantified subformulas. One contributing factor to this problem was that F7 communicates with the SMT solver Z3 using the Simplify format, while the more advanced SMT-Lib format would permit us to add sorting of logical variables, patterns to guide quantifier instantiation, and access to the array theory implemented in Z3 for a more efficient modelling of tables as arrays indexed by the primary key.

Given that our objective is to statically verify that transactional code contains enough checks to preserve the database invariants, we found that applying our approach interactively as we developed the transactional code helped us implement an exhaustive set of checks and made for a pleasant programming experience. Still, our approach leads to verbose code which, when verified, explicitly handles any possible transaction outcome.

7 Related Work

The idea of applying program verification to database updates goes back to pioneering work [16,9] advocating the use of Hoare logic or weakest preconditions to verify transactions.

Sheard and Stemple [26] describe a system for verifying database transactions in a dialect of ADA to ensure that if they are run atomically then they obey database constraints. The system uses higher order logic and an adaptation of the automated techniques of Boyer and Moore.

In the setting of object-oriented databases, Benzaken and Doucet [5] propose that the checking procedures invoked by triggers be automatically generated from high-level constraints, well-typed boolean expressions.

Benedikt, Griffin, and Libkin [3] consider the integrity maintenance problem, and study some theoretical properties of the weakest preconditions for a database transaction to succeed, where transactions and queries are specified directly in first-order logic and extensions. Wadler [30] describes a related practical system, Pdiff, for compiling transactions against a large database used to configure the Lucent 5ESS telephone switch. Consistency constraints on a database with nearly a thousand tables are expressed in C. Transactions in a functional language are input to Pdiff, which computes the weakest precondition which must hold to ensure the transaction preserves database integrity.

To the best of our knowledge, our approach to the problem is the first to be driven by concrete SQL table descriptions, or to be based on an interpretation of SQL queries as list processing and SQL constraints as refinement types, or to rely on SMT solvers.

A recent tool [12] analyzes ADO.NET applications (that is, C# programs that generate SQL commands using the ADO.NET libraries) for SQL injection, performance, and integrity vulnerabilities. The only integrity constraints they consider are check constraints (for instance, that a price is greater than zero); they do not consider primary key and foreign key constraints.

Malecha and others [18] use the Coq system to build a fully verified implementation of an in-memory SQL database, which parses SQL concrete syntax into syntax trees, maps to relational algebra, runs an optimizer and eventually a query. Their main concern is to verify the series of optimization steps needed for efficient execution. In contrast, our concern is with bugs in user transactions rather than in the database implementation. Still, our work in F# is not fully verified or certified, so for higher assurance it could be valuable to port our techniques to this system.

Ur/Web [10] is a web programming language with a rich dependent type system. Like our work, Ur/Web has a dependently typed embedding of SQL tables, and can detect typing errors in embedded queries. On the other hand, static checking that transactions preserve integrity is not an objective of the design; Ur/Web programs may result in a “fatal application error if the command fails, for instance, because a data integrity constraint is violated” (online manual, November 2010).

Refinement-type checkers with state are closely related to systems for Extended Static Checking such as ESC Java [15] and its descendants [2]. To the best of our knowledge, these systems have not previously been applied to verification of transactions, but we expect it would be possible.

8 Conclusion

We built a tool for SQL databases to allow transactions to be written in a functional language, and to be verified using an SMT-based refinement-type checker. On the basis of our implementation experience, we conclude that it is feasible to use static verification to tell whether transactions maintain database integrity.

In the future, we are interested to consider an alternative architecture in which our static analysis of queries is implemented in the style of proof-carrying code on the SQL server itself. Another potential line of work is to model database state within separation logic, and to appeal to its tools for reasoning about updates. Finally, it would be interesting to apply our techniques in the setting of software transactional memory, for example, on top of recent work on semantics for STM Haskell [7].

Acknowledgements. Discussions with Peter Buneman, Giorgio Ghelli, and Tim Griffin were useful. Karthik Bhargavan helped us with F7. David Naumann commented on an earlier version of this paper.

References

1. Baltopoulos, I.G., Borgström, J., Gordon, A.D.: Maintaining database integrity with refinement types. Technical Report MSR-TR-2011-51, Microsoft Research (2011)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Benedikt, M., Griffin, T., Libkin, L.: Verifiable properties of database transactions. *Information and Computation* 147(1), 57–88 (1998)
4. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffei, S.: Refinement types for secure implementations. In: Computer Security Foundations Symposium (CSF 2008), pp. 17–32. IEEE, Los Alamitos (2008)
5. Benzaken, V., Doucet, A.: Thémis: A database programming language handling integrity constraints. *VLDB Journal* 4, 493–517 (1995)
6. Bierman, G.M., Gordon, A.D., Hrițcu, C., Langworthy, D.: Semantic subtyping with an SMT solver. In: International Conference on Functional Programming (ICFP), pp. 105–116. ACM, New York (2010)
7. Borgström, J., Bhargavan, K., Gordon, A.D.: A compositional theory for STM Haskell. In: Haskell Symposium, pp. 69–80. ACM, New York (2009)
8. Borgström, J., Gordon, A.D., Pucella, R.: Roles, stacks, histories: A triple for Hoare. *Journal of Functional Programming* 21, 159–207 (2011); An abridged version of this article was published in A. W. Roscoe, Cliff B. Jones, Kenneth R. Wood (eds.), *Reflections on the Work of C.A.R. Hoare*, Springer London Ltd (2010)
9. Casanova, M.A., Bernstein, P.A.: A formal system for reasoning about programs accessing a relational database. *ACM Transactions on Programming Languages and Systems* 2(3), 386–414 (1980)
10. Chlipala, A.J.: Ur: statically-typed metaprogramming with type-level record computation. In: Programming Language Design and Implementation (PLDI), pp. 122–133. ACM, New York (2010)

11. Cooper, E., Lindley, S., Yallop, J.: Links: Web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)
12. Dasgupta, A., Narasayya, V.R., Syamala, M.: A static analysis framework for database applications. In: International Conference on Data Engineering (ICDE), pp. 1403–1414. IEEE Computer, Los Alamitos (2009)
13. Filliâtre, J.-C.: Proof of imperative programs in type theory. In: Altenkirch, T., Naraschewski, W., Reus, B. (eds.) TYPES 1998. LNCS, vol. 1657, pp. 78–92. Springer, Heidelberg (1999)
14. Flanagan, C.: Hybrid type checking. In: ACM Symposium on Principles of Programming Languages (POPL 2006), pp. 245–256 (2006)
15. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Programming Language Design and Implementation (PLDI), pp. 234–245 (2002)
16. Gardarin, G., Melkanoff, M.A.: Proving consistency of database transactions. In: Fifth International Conference on Very Large Data Bases, pp. 291–298. IEEE, Los Alamitos (1979)
17. Krishnamurthi, S., Hopkins, P.W., Mccarthy, J., Graunke, P.T., Pettyjohn, G., Felleisen, M.: Implementation and use of the PLT scheme web server. *Journal of Higher-Order and Symbolic Computing (HOSC)* 20(4), 431–460 (2007)
18. Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Principles of Programming Languages (POPL), pp. 237–248. ACM, New York (2010)
19. Meijer, E., Beckman, B., Bierman, G.M.: LINQ: reconciling object, relations and XML in the.NET framework. In: SIGMOD Conference, p. 706. ACM, New York (2006)
20. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: dependent types for imperative programs. In: International Conference on Functional Programming (ICFP 2008), pp. 229–240. ACM, New York (2008)
21. Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M.: An overview of the Scala programming language. Technical Report IC/2004/64, EPFL (2004)
22. Peyton Jones, S., Wadler, P.: Comprehensive comprehensions. In: Haskell 2007, pp. 61–72. ACM, New York (2007)
23. Ranise, S., Tinelli, C.: The SMT-LIB Standard: Version 1.2 (2006)
24. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: Programming Language Design and Implementation (PLDI), pp. 159–169. ACM, New York (2008)
25. Serrano, M., Galesio, E., Loitsch, F.: Hop: a language for programming the web 2.0. In: Object-oriented programming systems, languages, and applications (OOPSLA 2006), pp. 975–985. ACM, New York (2006)
26. Sheard, T., Stemple, D.: Automatic verification of database transaction safety. *ACM Transactions on Database Systems* 14(3), 322–368 (1989)
27. Swamy, N., Chen, J., Chugh, R.: Enforcing stateful authorization and information flow policies in FINE. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 529–549. Springer, Heidelberg (2010)
28. Syme, D., Granicz, A., Cisternino, A.: *Expert F#*. Apress (2007)
29. Wadler, P.: Comprehending monads. *Mathematical Structures in Computer Science* 2, 461–493 (1992)
30. Wadler, P.: Functional programming: An angry half-dozen. In: Cluet, S., Hull, R. (eds.) DBPL 1997. LNCS, vol. 1369, pp. 25–34. Springer, Heidelberg (1998)
31. Xi, H.: Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming* 17(2), 215–286 (2007)

Frequency Estimation of Virtual Call Targets for Object-Oriented Programs

Cheng Zhang¹, Hao Xu^{2,*}, Sai Zhang³, Jianjun Zhao^{1,2}, and Yuting Chen²

¹ Department of Computer Science and Engineering, Shanghai Jiao Tong University

² School of Software, Shanghai Jiao Tong University

{cheng.zhang.stap, steven_xu, zhao-jj, chenyt}@sjtu.edu.cn

³ Computer Science & Engineering Department, University of Washington
szhang@cs.washington.edu

Abstract. The information of execution frequencies of virtual call targets is valuable for program analyses and optimizations of object-oriented programs. However, to obtain this information, most of the existing approaches rely on dynamic profiling. They usually require running the programs with representative workloads, which are often absent in practice. Additionally, some kinds of programs are very sensitive to run-time disturbance, thus are generally not suitable for dynamic profiling. Therefore, a technique which can statically estimate the execution frequencies of virtual call targets will be very useful.

In this paper we propose an evidence-based approach to frequency estimation of virtual call targets. By applying machine learning algorithms on the data collected from a group of selected programs, our approach builds an estimation model to capture the relations between static features and run-time program behaviors. Then, for a new program, the approach estimates the relative frequency for each virtual call target by applying the model to the static features of the program. Once the model has been built, the estimation step is purely static, thus does not suffer the shortcomings of existing dynamic techniques. We have performed a number of experiments on real-world large-scale programs to evaluate our approach. The results show that our approach can estimate frequency distributions which are much more informative than the commonly used uniform distribution.

1 Introduction

Most of the object-oriented programming languages provide the virtual call mechanism to support polymorphism. While enhancing the modularity and extensibility in both design and implementation, virtual calls also complicate the static call graphs by adding extra branches at the call sites. As a result, call graph-based program analyses and optimizations may become less effective. In addition, virtual calls may cause significant performance overhead, because the

* He is currently a graduate student at Department of Computer Science, University of Southern California.

exact callees must be determined at run-time by selecting them from all the candidates based on the receiving objects (this process is often called *dynamic binding*). Therefore, it is important to resolve virtual calls at compile time or to obtain information about the execution frequencies of the targets (i.e., callees) for the unresolved virtual calls.

An empirical study [16] has shown that the distribution of execution frequencies of virtual call targets is highly peaked, that is, a small number of methods are frequently called in most of the run-time virtual calls. Thus it is rewarding to find the most frequently executed targets. Various dynamic profiling techniques are developed to explore the frequency distribution or relative frequencies of virtual call targets. Some of them [12] [16] [19] [14] [21] achieve high accuracy with relatively low overhead. However, in order to generate useful profiles, most of the dynamic techniques require driving programs with representative workloads, which are often absent, especially for newly developed programs. Moreover, dynamic techniques are usually intrusive in that they have to instrument programs to collect information, whereas some kinds of programs (e.g., multi-thread programs) may be extremely sensitive to run-time disturbance. In these cases, a static technique with acceptable accuracy could be a preferable alternative.

In this paper, we propose *Festival*, an evidence-based approach to **frequency estimation of virtual call targets**. The underlying assumption of Festival is that developers' design intentions, which cause the imbalance of usage of different virtual call targets, can be revealed by examining a group of static program-based features. Festival consists of two phases: 1) model building and 2) estimation. In the model building phase, Festival selects a set of existing programs with representative workloads, extracts some static features, and runs the programs to get the dynamic profiles for their virtual call targets. Based on the collected data, Festival uses machine learning algorithms to discover the relationship between the execution frequencies and the static features. The relationship is represented as an artificial neural network. As a prerequisite for model building, we assume that the programs and their representative workloads are available. This assumption is reasonable, because there exist numerous object-oriented programs, which have been used in practice for years. The accumulated workloads for such programs are probably representative. In the estimation phase, for a new program, Festival extracts the same set of features from it and uses the model to estimate the relative frequencies of the virtual call targets in the program. It is worth noting that, once the model has been built, the estimation phase is purely static. We have implemented a prototype of Festival and performed a set of experiments on the DaCapo benchmark suite [7]. The experimental results show that the estimated frequency distributions are significantly more informative than the uniform distribution which is commonly used in static analyses.

The main contributions of this work can be summarized as:

1. Festival, the first evidence-based approach we are aware of to estimate frequencies of virtual call targets for object-oriented programs. It can be a good complement to existing dynamic techniques. As will be discussed in Section 3, a variety of client applications may benefit from our approach.

2. An evaluation conducted to validate the effectiveness of our Festival approach. It consists of a comprehensive group of experiments, which show the estimation performance of Festival from various aspects.

The rest of this paper is organized as follows. Section 2 uses an example to give a first impression of the static features. Section 3 discusses a number of potential applications of our Festival approach. Section 4 describes the technical details of the approach. Section 5 shows the experimental results. Section 6 compares Festival with related work and Section 7 concludes the paper and describes our future work.

2 Motivating Example

In this section we use a real-world example to illustrate some of the features used in our approach. The features are program-based and easy to extract using static analysis. Nevertheless, we believe that they are related to the run-time execution frequency of virtual call targets.

The code segments shown in Figure 1 are excerpted from ANTLR (version 2.7.2) [1], a parser generator written in Java. From the code, we can see that class `BlockContext` contains three fields and three methods, while its subclass `TreeBlockContext` has only one field and one method, `addAlternativeElement`, which overrides the implementation provided by `BlockContext`. The virtual call of interest is at line 6. The method `context` (whose definition is omitted for brevity) has a return type `BlockContext`. Thus the virtual call has two possible targets: one is the method `addAlternativeElement` defined in `BlockContext` and the other is the one defined in `TreeBlockContext`. By running ANTLR using the workload provided in DaCapo benchmark (version 2006-10-MR), we obtained the dynamic profiles of these two targets and found that the execution frequency of the method defined in `BlockContext` is about ten times higher than that of the method defined in `TreeBlockContext`. But what if we cannot run the program, say, because the workload is unavailable? Can we make a good guess at the relative frequencies of these two targets?

If we analyze the program source code, some informative evidences can be discovered. First, `TreeBlockContext` is a subclass of `BlockContext`. As a general rule of object-oriented design, the subclass (i.e., `TreeBlockContext`) is a specialized version of the superclass (i.e., `BlockContext`). Second, because the method `addAlternativeElement` has a concrete implementation in `BlockContext` rather than being abstract, it is probably designed to provide common functionalities, while the method in `TreeBlockContext` is designed for special cases. Third, if we explore the calling relations between relevant methods, we will find that the method defined in `TreeBlockContext` calls its super implementation (the call is at line 36 in Figure 1). It indicates that `TreeBlockContext` delegates a part of its responsibility to its superclass. At last, there are three fields and three methods defined in class `BlockContext`, while class `TreeBlockContext` has only one field and one method. The fact that `BlockContext` has higher complexity may also show its relative importance. Based on these evidences we are likely to consider

```

1  public class MakeGrammar
2      extends DefineGrammarSymbols {
3      protected void addElementToCurrentAlt (
4          AlternativeElement e) {
5          ...
6          context().addAlternativeElement(e);
7      }
8
9  class BlockContext {
10     AlternativeBlock block;
11     int altNum;
12     BlockEndElement blockEnd;
13
14     public void addAlternativeElement(
15         AlternativeElement e) {
16         currentAlt().addElement(e);
17     }
18
19     public Alternative currentAlt() { ... }
20     public AlternativeElement
21         currentElement() { ... }
22 }
23
24 class TreeBlockContext
25     extends BlockContext {
26     protected boolean
27         nextElementIsRoot = true;
28
29     public void addAlternativeElement(
30         AlternativeElement e) {
31         TreeElement tree=(TreeElement)block;
32         if (nextElementIsRoot) {
33             tree.root=(GrammarAtom)e;
34             nextElementIsRoot = false;
35         }else {
36             super.addAlternativeElement(e);
37         }
38     }
39 }

```

Fig. 1. Code segments from ANTLR 2.7.2

the method in `BlockContext` as the major one, and thus correctly predict a higher frequency for it.

The example presents the intuition that design intentions can be revealed by analyzing static features. This strongly motivates us to use such kind of features to estimate relative frequencies of virtual call targets. However, it is still challenging to tell how a specific feature may indicate the frequencies. Moreover, when there is a large amount of feature data extracted (especially for large programs),

different features may lead to contradictory judgements in some cases. Consequently, the problem of how to make optimized estimations based on such kind of features motivates us to leverage the power of machine learning techniques, which are devised to discover useful knowledge from data.

3 Potential Applications

Various techniques dependent on frequency information of virtual call targets may benefit from Festival. On one hand, traditional profile-guided techniques can use the estimated profiles when dynamic profiles are unavailable. It makes them applicable in more situations. On the other hand, static techniques may achieve better performance by using more accurate information.

Program optimizations usually use dynamic profiles to help make economic optimization decisions. Nevertheless, when dynamic profiling is inappropriate, Festival can be a good substitution. Sometimes it may be integrated into the optimization process more conveniently than dynamic profilers. For instance, as shown in Figure 2, when performing the class test-based optimization [16], the compiler can insert a test for the dominant class (i.e., the class which defines the most frequently executed target method) and statically determine the target method in the successful branch. Since the test is mostly successful at run-time, the overhead of dynamic binding can be reduced. If Festival is used to identify the dominant class, this optimization can be performed without running the program.

```

Before optimization :
TypeA a = ...;
a.method();

After optimization :
TypeA a = ...;
if ( a instanceof DominantSubtypeOfA ) {
    ((DominantSubtypeOfA) a).method();
} else {
    a.method();
}

```

Fig. 2. An example of code optimization

Another application of Festival may be the probabilistic program analyses. Besides computing the *must* or *may* behaviors of programs, probabilistic program analyses [5] [18] also show the likelihood of each *may* behavior's occurrence. For example, the probabilistic points-to analysis [18] assigns a frequency distribution to each points-to set indicating which memory locations are more likely to be the target of the pointer. Since the existing work focuses on C language, it performs analysis based on control-flow graphs which do not involve virtual calls. If the work is extended to handle object-oriented languages (e.g., Java),

it can use Festival, at the beginning of its analysis, to allocate probabilities to virtual call targets instead of assuming a uniform frequency distribution.

For static bug finding tools (e.g., Findbugs [17]), high false positive rate is a major obstacle to their applications. Thus alert ranking methods are introduced to reduce the effort for finding real warnings. Some methods (e.g., [9]) rank alerts in terms of the execution likelihood of program elements. They calculate the likelihood by propagating probabilities along the edges of control-flow graphs. Using the frequency information of virtual call targets, the propagation may be more accurate when it has to branch at virtual call sites. Then better results of alert ranking may be obtained.

In summary Festival can be useful to call graph-based analyses and optimizations that deal with object-oriented programs. While the frequency estimation only needs static features of programs, a model must be built beforehand. In the next section, we will describe the technical details.

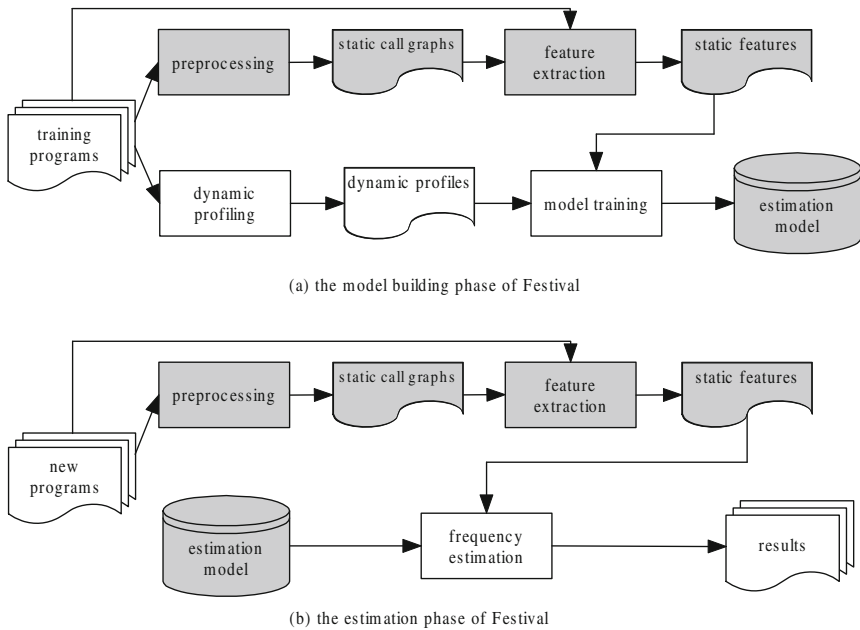


Fig. 3. The architecture of Festival which consists of two phases. In the first phase, static features and dynamic profiles are used to build the estimation model. In the second phase, only static features are extracted from new programs to perform frequency estimation. The shaded elements stand for the steps or entities involved in both phases.

4 Approach

Figure 3 shows the architecture of Festival, which consists of two phases: a) model building and b) estimation. At the beginning of the model building phase, a set of selected programs are preprocessed to construct their static call graphs. Then a

group of static features are extracted based on the call graphs and the programs’ source code. Meanwhile dynamic profiling is performed to get dynamic profiles for these programs. In the end of this phase, the estimation model is built based on the static features and dynamic profiles using machine learning algorithms. Once the model has been built, it will be repeatedly used afterwards. In the estimation phase, static features are extracted, in the same way, from the new programs which are not used in model building. Then the relative frequencies for virtual call targets are estimated by applying the estimation model to the features.

4.1 Preprocessing

The purpose of the preprocessing step is to construct precise static call graphs for feature extraction. In this step, we use the points-to analysis provided by the Spark framework [20] to compute the types of objects that may be referenced by each variable. Based on the type information of the receiver variable, the set of possible targets for each virtual call can be computed more precisely than traditional algorithms used for call graph construction (e.g., CHA [13]). As a result, a number of single-target virtual calls are resolved before the subsequent steps. The preprocessing step enables our approach to focus on real multi-target virtual calls¹ in order to handle large-scale programs. Hereafter the term “virtual call” means multi-target virtual calls, unless we explicitly state that a virtual call is single-target.

4.2 Static Feature Set

A static feature can be viewed as a specific measure used to capture one characteristic of a virtual call target. Thus different targets may have different values for the same feature. We use 14 static features (as shown in Table 1) to characterize virtual call targets from various aspects, including type hierarchy, calling relation, naming style, program complexity, etc.

Type hierarchy features. Features 1 to 5 are designed to represent information about the type hierarchy of the classes in which the target methods are defined. Hereafter we call such a class as a *target class* and all the target classes of a virtual call comprise the *target class set* (of that virtual call). For a specific virtual call site, the receiver variable has its explicit type (we call it the *called type*) and each target class must be either the called type itself or a subtype of the called type. Figure 4 shows the class diagram of an example for illustrating the five features related to type hierarchy. As shown in the figure, **A** is an interface which has three implementers **B**, **C**, and **D**. Class **D** extends class **C** and class **C** implements interface **F** which has no inheritance relationship with interface **A**. In addition, classes **B**, **C**, and **D** all have their own implementations of method **m**. Suppose that at a call site the method **m** is called on a variable of type **A**. In this case, the called type is **A** and the target class set is $TC(\mathbf{A}) = \{\mathbf{B}, \mathbf{C}, \mathbf{D}\}$.

¹ Since the problem of points-to analysis is undecidable in general, some real single-target virtual calls may still be regarded as multi-target.

Table 1. Static features used in Festival

Number	Feature Name	Feature Description
1	Type Distance	the number of levels of subtyping from the called type to the target class
2	Subclass in TC	the number of subclasses of the target class in the target class set
3	Superclass in TC	the number of superclasses of the target class in the target class set
4	Subtree Size	the number of subclasses of the target class in the whole program
5	Number of Ancestors	the number of supertypes of the target class in the whole program (except for library types)
6	Does Call Super	whether the target method calls its super implementation (yes or no)
7	Number of Callers	the number of methods which call the target method
8	Package Depth	the depth of the package of the target class
9	Name Similarity	the number of classes whose names are similar to that of the target class
10	Number of Methods	the number of methods defined in the target class
11	Number of Fields	the number of fields defined in the target class
12	Is Abstract	whether the target class is abstract (yes or no)
13	Is Anonymous	whether the target class is anonymous (yes or no)
14	Access Modifier	the access modifier of the target class (public, protected, private, default)

Feature 1 (type distance) measures the distance between a target class and the called type, that is, it records the number of edges on the path between the corresponding nodes on the class diagram. If there is more than one path, the shortest path will be used for this feature. In the example, the feature value of both **B** and **C** is 1, while that of **D** is 2. A possible heuristic may be that the target method whose class has shorter distance to the called type will have higher execution frequency, since the target class is more general and likely to be designed for handling common cases. Features 2 (subclass in TC) and 3 (superclass in TC) encode the inheritance relations among the target classes **in the same target class set**. For example, **C** has the values 1 and 0 for feature 2 and feature 3, respectively, because **C** is the superclass of **D** which also belongs to $TC(\mathbf{A})$ and **C** has no superclass in $TC(\mathbf{A})$. Meanwhile, as there is no subclass or superclass of **B** in $TC(\mathbf{A})$, **B** has the value 0 for both of the features. Sometimes target classes are “parallel” to each other, that is, there are no inheritance relations between them, such as **B** and **C**. Thus, for each target class, we use features 4 (subtree size) and 5 (number of ancestors) to count the numbers of its subclasses and supertypes in the whole program so that the relative importance of the “parallel” target classes can be characterized. In the example, the values of features 4 and 5 for **B** are 0 and 1, while they are 1 and 2 for **C**. Note that library types (e.g., `java.lang.Object`) are not counted in these two features.

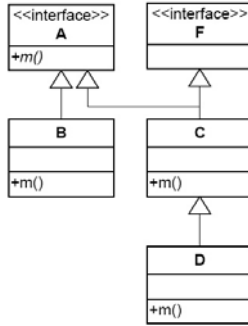


Fig. 4. An example class diagram

Call graph features. Similar to type hierarchy, calling relations may also provide hints to design intentions. Thus two of the features are based on call graph. Feature 6 (does call super) shows whether the target method calls the implementation of the superclass. If the target method just overrides its super implementation, it is not so perceivable which of the implementations is designed to take more responsibility. On the contrary, as discussed in Section 2, if the target method calls its super implementation, it is very likely that it delegates a part or all of its job to the callee. As a result, the super implementation may have much higher execution frequency. The other call graph feature, feature 7 (number of callers), tries to indicate the target method's popularity in the scope of the whole program. The intuition is that the more callers a target method has, the more popular it is. High popularity means high probability for the method to be a central part of the program, which may result in high execution frequency of the method in run-time virtual calls. Note that feature 7 counts in all the methods that explicitly call the virtual call target. In other words, its value is equal to the number of incoming edges of the virtual call target in the static call graph. Among these call graph edges, some represent resolved (single-target) virtual calls, while others represent unresolved (multi-target) virtual calls. In general, we focus on the latter, that is, we record (during dynamic profiling) and estimate execution frequencies only at the unresolved call sites. However, in feature 7, we take into account both resolved and unresolved virtual calls to a target method in order to characterize its popularity.

Naming style features. In practice several naming conventions are used to organize program elements in terms of their functionalities. A typical example is the name space mechanism provided in various programming languages. In Java, where name spaces are specified by package names, the depth of package may indicate the specialty of functionality of the classes in the package. Therefore, feature 8 (package depth) is designed to represent this characteristic. From another viewpoint on naming style, feature 9 (name similarity) counts the number of classes whose names are similar to that of the target class. If some classes have similar names, they may collaborate with each other to accomplish

the same task. The more classes involved, the more important the task may be. Again the relative importance can be used to estimate execution frequencies. When computing the value of feature 9 for a given target class, we find similar class names in the following steps:

1. Find the longest common suffix of the names of the classes in the target class set.
2. Identify the special prefix for the given target class by removing the longest common suffix from its name.
3. Within the whole program, the class names that begin with the special prefix are considered as similar to the name of the target class.

Suppose, for a specific virtual call, there are two target classes `WalkingAction` and `DrivingAction`. Then the longest common suffix is “Action” and the special prefix is “Walking” for `WalkingAction`. Therefore, the number of classes (all over the program) whose names begin with “Walking” is recorded as the value of feature 9 for the target method defined in `WalkingAction`. Note that library classes are not taken into consideration during the computation, because their names usually do not indicate the design of application classes.

Complexity features. In some cases, program complexity metrics can be used to represent the importance of a program element. Currently we use two simple metrics, features 10 (number of methods) and 11 (number of fields), to measure the complexity of the target classes, because these two features are easy to extract and have satisfactory predictive power. It is worth noting that we do not add complexity of inner classes or anonymous classes to their outer classes, because we believe they are less coherent to the outer classes than the member methods and fields. We have also tried other common metrics (e.g., line of code), but found them less indicative. In our future work, we are planning to investigate some more complex metrics, such as the depth of nesting loops and the number of program paths.

Other features. Features 12 (is abstract), 13 (is anonymous), and 14 (access modifier) are mainly about surface properties of the target class. These features are used to capture design intentions from aspects other than the aforementioned ones. For example, a class is defined as abstract (instead of an interface) probably means that it provides some method implementations that will be reused by its subclasses. Thus feature 12 may be useful when we investigate the frequencies of the implemented methods.

In general the static features are selected to represent evidences which are supposed to be indicative of execution frequencies. A variety of heuristics may be proposed based on these features. Nevertheless, counterexamples may be found against each heuristic by checking the dynamic profiles, and there may also be contradictions between indications of different features. Thus we use machine learning techniques to analyze the feature data and discover relatively consistent knowledge in order to make efficient frequency estimation.

4.3 Estimation Model

We formulate the frequency estimation problem as a supervised classification problem [24] in machine learning. The task of classification is to determine which category (usually called *class*) an instance belongs to, based on some observable features of the instance and a model representing the existing knowledge. A classification problem is said to be supervised when the model is trained (i.e., built) using instances (called *training instances*) whose classes have already been specified. A typical example of supervised classification is to predict the weather condition (sunny, cloudy, or rainy) of a specific day based on some measures (temperature, humidity, etc.) of that day and a forecast model derived from historical weather record. In Festival, we build a model to classify each virtual call target as *frequent* or *infrequent* and use the predicted probability of being frequent as the estimated frequency.

In essence a model is a parameterized function, which represents the relations between its input and output. In Festival, for a specific virtual call target, the input of the estimation model is the values of the target's static features, and the output is the estimated frequency for the target. Therefore, the estimation model of Festival correlates static features with actual frequencies, providing a way to estimate unknown frequencies of new targets on the basis of the targets' static features. A key assumption of Festival is that the relations between static features and dynamic behaviors of virtual calls are stable across different programs. In other words, we can build a model based on some programs and use it to estimate frequencies for others.

In order to build the estimation model, we first select a set of programs (called *training programs*) and extract their values for the static features described in Section 4.2. Then we instrument the training programs at each virtual call site and run them with their representative workloads. The run-time execution count of each target is recorded during the execution [2]. Since the workloads are representative, the execution counts can be used as the real execution frequencies of the targets. When both static features and dynamic profiles have been obtained for training programs, we are ready to build the model.

During model building, each virtual call target corresponds to an instance which is represented as a vector $\langle f_1, f_2, f_3, \dots, f_{14}, c \rangle$, where f_i is the value of the i th feature and c is the recorded execution count. We have to process the instance data to make them fit for our approach. Because targets from different virtual calls are used together for model training, their feature values should be measured relatively within each virtual call. Thus we normalize the feature values within all the targets of the same virtual call. For example, if the values of feature 1 for three targets (of a specific virtual call) are f_1^x , f_1^y , and f_1^z , then they will be normalized as $\frac{f_1^x}{\max\{f_1^x, f_1^y, f_1^z\}}$, $\frac{f_1^y}{\max\{f_1^x, f_1^y, f_1^z\}}$, and $\frac{f_1^z}{\max\{f_1^x, f_1^y, f_1^z\}}$, respectively. Moreover, to specify the class of each instance, we order the targets of each

² More specifically, what we record is the number of times a method becomes the actual target of its corresponding virtual call, rather than the total number of times a method is called.

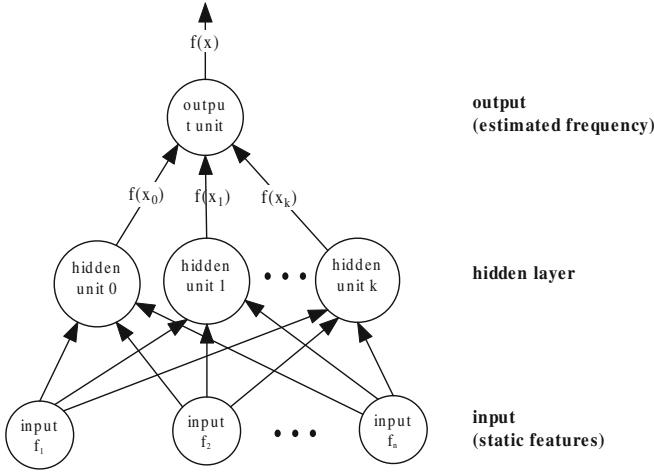


Fig. 5. The multilayer perceptron used in frequency estimation. Each input f_i corresponds to a static feature of a virtual call target and the output $f(x)$ is used as the estimated frequency of the target.

virtual call with respect to their execution counts in a descending order. Then we assign class **1** (means “frequent”) to the top 20% targets and class **0** (means “infrequent”) to the others. When a virtual call has less than 10 targets, we assign class **1** to the top one target and class **0** to the others. After the data processing, the instances are transformed into the form $\langle f'_1, f'_2, f'_3, \dots, f'_{14}, L \rangle$, where $L \in \{1, 0\}$ and f'_i is the normalized value of f_i . Then we use the processed instances to train a multilayer perceptron [24], which will be used as our estimation model.

A multilayer perceptron is a kind of artificial neural network which is illustrated in Figure 5. In the multilayer perceptron, the output of a node i in the hidden layer is described by

$$f(x_i) = \frac{1}{1 + e^{-x_i}}$$

where x_i is the weighted sum of its inputs, that is, $x_i = w_{i0} + w_{i1}f_1 + w_{i2}f_2 + \dots + w_{in}f_n$. Using the same function, the final output $f(x)$ is computed by taking the outputs of the hidden nodes as its inputs (i.e., $x = w_0 + w_1f(x_1) + w_2f(x_2) + \dots + w_kf(x_k)$). The final output represents the probability for an instance to be of a certain class (e.g., class **1**). During model training the weights are so computed that the total squared error of the output is minimized. The squared error of a single instance i is described by

$$error_i = \frac{1}{2}(L_i - f(x)_i)^2$$

where L_i is the class of i and $f(x)_i$ is the model’s output for i . If there are totally N instances used in model training, then the total squared error is $\sum_{i=1}^N error_i$. When the weights have been established, the model training is finished.

During frequency estimation, a new target, whose frequency is to be estimated, is viewed as a new instance whose class is unknown. Thus we encode it as a feature vector $\langle f'_1, f'_2, f'_3, \dots, f'_{14} \rangle$ and feed the vector as the input to the multilayer perceptron. In the end, the outputted probability is considered as the estimated frequency of the target.

5 Evaluation

To evaluate our Festival approach, we have implemented a prototype on the basis of the Soot framework [4] and the Weka toolkit [24]. Soot is used to construct static call graphs³ and extract static features, while Weka is used for machine learning. We have performed a set of experiments on the implementation prototype. Through the experiments we try to answer the following research questions:

- RQ1: What is the estimation performance of Festival?
- RQ2: Is Festival applicable to various programs?
- RQ3: What is the predictive power of each feature?

5.1 Experimental Design

Subject programs. In the experiments we use 11 programs from the DaCapo benchmark suite (version 9.12-bach) as the subject programs, because DaCapo provides comprehensive workloads for each program. Moreover, as the benchmark suite is originally designed for Java runtime and compiler research (especially for performance research), we believe the workloads are representative. Table 2 shows the basic characteristics of the subject programs. The column #M shows the number of methods that are included in the static call graph of each program, and the column #VC shows the number of virtual calls that have multiple targets as identified by the points-to analysis. From the columns LOC and Description, we can see that the subject programs are medium-to-large real-world programs used in various application domains. Thus they are quite suitable for our study on the research questions, especially RQ2. Note that we have not selected three programs (namely jython, tradebeans, and tradesoap) from DaCapo. Because tradebeans and tradesoap involve too much multi-threading, it is difficult for us to obtain their representative dynamic profiles. As for jython, we failed to finish the instrumentation (for dynamic profiling) within tens of hours. Although these three programs have been left out in the experiments, we believe that the selected 11 subject programs are sufficient to validate our Festival approach. Moreover, the lack of dynamic profiles does not indicate that these programs cannot be estimated by Festival. It just prevents us from using the programs for model training and evaluating Festival’s performance on them.

Model training scheme. In the experiments, multilayer perceptron is used as the machine learning model in a way which is described in Section 4.3. During

³ The aforementioned Spark framework is a building block of Soot. We use Soot 2.4.0 together with TamiFlex [8] to build call graphs that include method calls via reflection.

Table 2. Subject programs from DaCapo-9.12-bach (LOC is measured using cloc version 1.51)

Name	Description	LOC	#M	#VC
avrora	simulation and analysis tool	68864	2439	34
batik	SVG toolkit	171484	5365	321
eclipse	non-gui part of Eclipse IDE	887336	18632	3007
fop	PDF file generator	96087	5503	682
h2	in-memory database	78124	3902	250
luindex	text indexing	36099	1546	107
lusearch	text searching	41153	1201	70
pmd	code analyzer	49610	4741	110
sunflow	rendering system	21960	1096	27
tomcat	web application server	158658	11554	4417
xalan	XSTL processor	172300	4366	547

Table 3. Numbers of virtual calls categorized by target number

Name	2	3	4	5	6 ~ 10	≤ 10	> 10
avrora	10	3	3	0	3	19	4
batik	52	40	3	5	7	107	12
eclipse	626	128	121	73	119	1067	380
fop	60	10	3	3	4	80	26
h2	10	4	3	15	13	45	42
luindex	31	5	5	0	1	42	0
lusearch	16	6	0	1	0	23	0
pmd	46	1	1	1	3	52	2
sunflow	11	5	4	0	0	20	0
tomcat	2309	59	17	32	62	2479	75
xalan	25	16	0	2	15	58	20
total	3196	277	160	132	227	3992	561

model training, we take the leave-one-out strategy. That is, while evaluating the performance of Festival on one specific subject program, we use the other ten subject programs as training programs. In this way, the virtual call targets, whose frequencies are estimated by the model, are never used to train that model.

Another special strategy we take for model training is to use the data only from the virtual calls whose numbers of targets are less than or equal to **10**. It is mainly due to the fact that virtual calls with too many targets may affect the training data drastically, whereas the current static features can hardly represent the information embedded in such kind of virtual calls (This limitation of Festival will be discussed in Section 5.6). Nevertheless, as shown in Table 3, most virtual calls in the subject programs have relatively small numbers of targets⁴.

⁴ Table 3 shows only the virtual calls that are executed during dynamic profiling, thus the total number of virtual calls is smaller than that shown in Table 2.

Therefore, our model training scheme takes into account the vast majority of the cases. Note that we obtained similar experimental results when we limited the number of targets to 5 and 15.

Platform and runtime. The experiments have been conducted on a Linux server, which has a 2.33GHz quad-core CPU and 16GB main memory. We use IBM J9 VM for feature extraction and Sun HotSpot VM for dynamic profiling. Although DaCapo provides workloads of different sizes, including small, default, large, and huge, we only use the large workloads for dynamic profiling. Because huge workloads are not available for most benchmarks, and small and default workloads are generally less representative than large ones⁵. Table 4 shows the runtime of each step in Festival, including preprocessing (PRE), feature extraction (FE), instrumentation (INS), execution (EXE), model training (MT), and frequency estimation (EST). We can see that the time cost is reasonable even for large-scale programs.

Table 4. Runtime of each step in Festival (EST is measured by second and others use the format of h:mm:ss)

Name	PRE	FE	INS	EXE	MT	EST
avrora	1:27	1:27	0:35	8:00	0:36	0.18
batik	8:21	8:22	6:46	0:09	0:34	0.16
eclipse	14:41	12:05	3:50:47	5:33	0:24	0.56
fop	14:12	14:16	12:07	0:04	0:36	0.11
h2	2:21	2:11	8:21	1:47	0:35	0.15
luindex	1:31	1:31	0:28	0:08	0:34	0.09
lusearch	1:20	1:20	0:21	0:54	0:35	0.06
pmd	2:18	2:15	2:54	0:21	0:34	0.11
sunflow	4:19	4:22	0:27	29:17	0:36	0.08
tomcat	15:35	15:32	1:19:25	0:24	0:17	0.71
xalan	2:13	2:15	3:16	4:23	0:33	0.09

5.2 Rank Correlation Analysis

In this experiment, we evaluate the agreement between the estimated and real frequency distributions. Specifically, for each virtual call, we first rank its targets according to their estimated frequencies and dynamic profiles, respectively. Then we measure the correlation between these two ranks by computing their Kendall tau distance [3]. Conceptually Kendall tau distance represents the similarity between two ordered lists by counting the number of swaps needed to reorder one list into the same order with the other. The normalized value of Kendall tau distance lies in the interval $[0, 1]$, where low distance value indicates high agreement. The average normalized value of Kendall tau distance between a list and its random permutation is 0.5. Because in this experiment random permutation corresponds to the uniform frequency distribution, we use 0.5 as the baseline.

⁵ Since fop and luindex do not have large workloads, we use default ones instead.

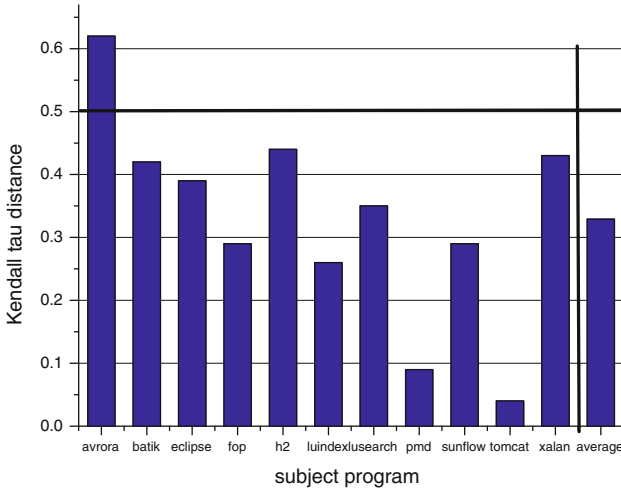


Fig. 6. Kendall tau distance between estimated and real distributions

Figure 6 shows the average normalized Kendall tau distance for virtual calls in each subject program as well as the overall average distance. Except for avrora, all the subject programs have a distance less than 0.5 and the overall average distance is 0.33. It indicates that the estimated frequency distributions can reflect the real distributions much better than the uniform distribution. By inspecting the relevant data of avrora, we found that some static features that can characterize avrora well (i.e., package depth and name similarity) have relatively low predictive power in the estimation model trained using the other subject programs. It might be due to the different design styles between avrora and other subject programs.

5.3 Top Target Prediction

According to the study by Grove et al. [16], one or two “hottest” targets usually take up most of the execution count of a virtual call. Therefore, it is meaningful to evaluate Festival’s ability to predict the top targets. As the virtual calls that we study have at most 10 targets, we focus on the top **one** target of each virtual call. The measure is straightforward: for a specific virtual call, if the estimated top target actually has the largest execution count in the dynamic profile, we score the prediction as 1; otherwise the score is 0. Figures 7 and 8 show the average scores of top target prediction based on Festival and the uniform distribution, where the scores are categorized by the number of targets and the subject program, respectively.

As shown in Figure 7, we get a mixed result in top target prediction: for some target numbers, Festival significantly outperforms uniform estimation, whereas it has much worse performance for others. It is difficult for Festival to constantly

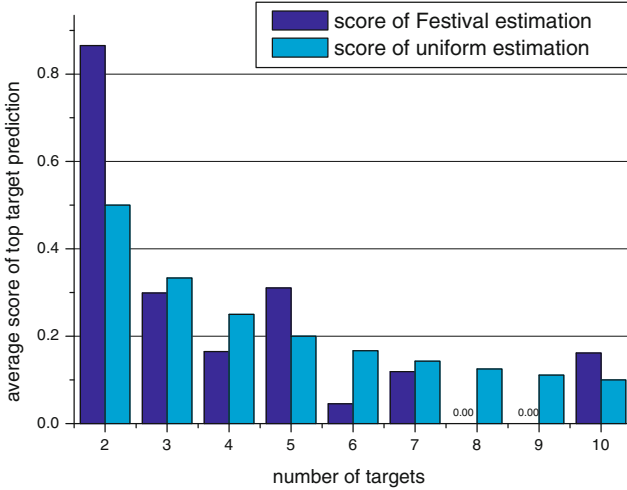


Fig. 7. Average scores of top target prediction categorized by number of targets

predict the top one target, especially when the number of targets is relatively large. However, as shown in Table 3, over 80% of the virtual calls (used for evaluation) have two targets. Therefore, the average score of 0.87 for two-target virtual calls can be viewed as more important than the other scores.

Figure 8 shows Festival’s performance for top target prediction from another point of view. When the scores are averaged within each subject program, Festival mostly outperforms uniform estimation. To be more detailed, for each subject program, the average scores of Festival and uniform estimation are calculated by $\frac{S_{total}}{N_{vc}}$ and $\frac{1}{A_{tgt}}$, respectively, where S_{total} stands for the total score of all virtual calls, N_{vc} stands for the number of virtual calls, and A_{tgt} stands for the average number of targets for each virtual call. Conceptually, $\frac{S_{total}}{N_{vc}}$ represents the likelihood for Festival to score 1 for each virtual call, and $\frac{1}{A_{tgt}}$ represents the likelihood to randomly predict the top target of a virtual call. Therefore, it is reasonable to compare them with each other. As for avrora, Festival does not perform well, which is probably due to the same reason as discussed in Section 5.2

5.4 Weight Matching Analysis

Besides the order of virtual call targets, the quantity of the estimated frequency may also be useful in some quantitative analyses. In this experiment, we use weight matching score [23] to measure the estimation performance of Festival in this aspect. For example, Table 5 shows five virtual call targets, along with their (normalized) estimated and real profiles. The two target lists are ordered by the estimated and real profiles, respectively.

In computing the weight matching score, a cut-off n is specified at first. Then we calculate the sum of **real profile values** for the top n targets in the

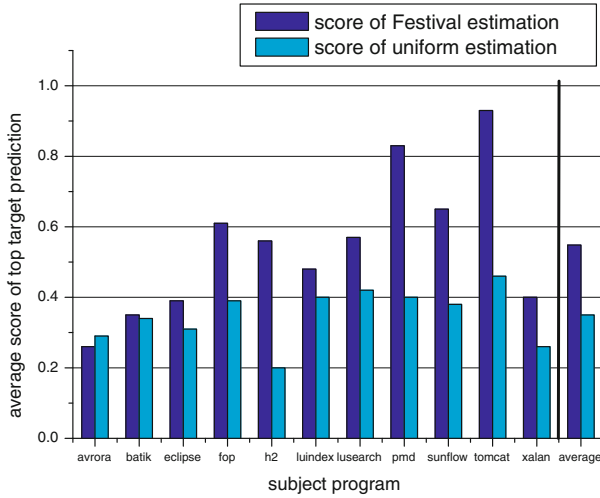


Fig. 8. Average scores of top target prediction categorized by subject program

estimated list as well as the sum for the top n in the **real** list (noted as Sum_e and Sum_r , respectively). The weight matching score is the ratio Sum_e/Sum_r . The perfect estimation has a score of 1, and the closer to 1, the better the estimation is. In the example, for $n = 2$, the weight matching score is $0.85/0.95$.

Table 5. An example for computing weight matching score

Estimated	Target	Real	Target
0.50	A	0.80	A
0.20	B	0.15	C
0.10	C	0.05	B
0.10	D	0.00	D
0.10	E	0.00	E

In this experiment, we calculate the scores with the cut-off $n = 1$. As shown in Figure 9, the average weight matching score of Festival is about 59%. For the subject programs, the top target averagely takes up 91% of the execution count in terms of the dynamic profiles. Thus Festival assigns more than a half (i.e., 54%) of the execution count to the estimated top target in average.

5.5 Predictive Power Analysis

This experiment is designed for investigating the relative predictive power of each static feature. To this end, we first build the estimation model based on each single feature instead of the whole feature set. Then we compute the Kendall tau

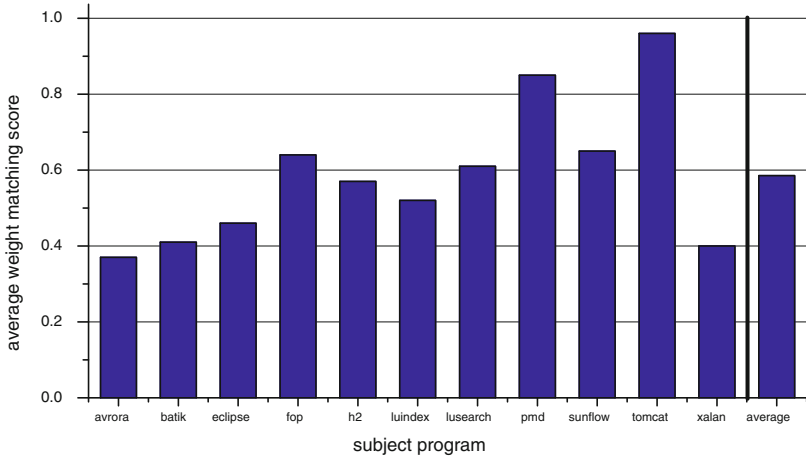


Fig. 9. Average weight matching scores

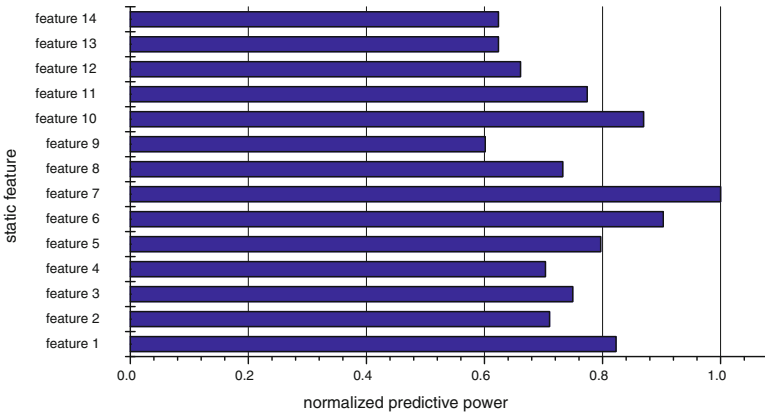


Fig. 10. Predictive power of static features

distance, in the same way as the first experiment, to measure the performance of the model, which in turn represents the predictive power of the feature.

Figure 10 shows the results, in which we use the normalized reciprocal of the Kendall tau distance to represent the predictive power. Thus the features that have larger values are more predictive than the others. In the experiment, the most predictive two features are both call graph-based, while the five features based on type hierarchy are seemingly less useful. However, as discussed in Section 2, feature 6 is closely related to features 2 and 3. Thus the correlations between call graph and type hierarchy may have large impact on the estimation. Similar to those based on type hierarchy, the complexity-based features

have moderate predictive power. In contrast, the surface features of target class generally have the lowest power. They might be too simple to capture sufficient design intentions. For naming style features, the depth of package is more informative than the name similarity. However, by analyzing the source code, we have found that the latter may become more predictive, if we have a better algorithm for computing name similarity. We are planning to improve this feature in our future work.

5.6 Discussion

The first three experiments have evaluated the estimation performance of Festival with respect to different measures. The experimental results give satisfactory answers to the research questions *RQ1* and *RQ2*. In addition, the analysis based on the fourth experiment presents an answer to *RQ3*. In summary, Festival can indeed provide useful information of execution frequency for virtual call targets.

Currently Festival has a limitation that it cannot provide useful estimation for virtual calls that have too many targets. A typical case is the *visitor* design pattern [15], which usually involves complex type hierarchies. For example, the eclipse JDT compiler API uses visitor pattern to process AST. Consequently, the class `ASTNode` has more than 90 subclasses, most of which have implemented the `accept` method. It is really difficult to estimate frequencies for so many implementations of the `accept` method. Another difficulty may stem from the fact that such kind of type hierarchies usually indicate complicated design intentions that can hardly be captured by our current static features.

5.7 Threats to Validity

One threat to the validity of the experiments is that we evaluate the accuracy of our approach by comparing the estimated frequencies with dynamic profiles. If the workloads are not representative, the comparison may lead to skewed results. To alleviate this problem, we choose subject programs from the DaCapo benchmark suite. Another threat is overfitting which means the machine learning model fits too well to the training data and has poor predictive performance on unseen test data. In the experiments we always leave out the program to be estimated and train the model using all the other programs. We believe this kind of cross-validation can avoid the threat of overfitting. Finally, the static features we use may not be comprehensive enough to capture all the valuable information. In fact we have studied over 20 features and selected the most informative 14 of them to build the estimation model.

6 Related Work

Machine learning is a powerful tool for predicting program behaviors. Calder et al. [11] proposed a branch prediction technique using decision trees and neural nets and coined the term *evidence-based static prediction* or *ESP*. Our approach

has a similar architecture to their work. However, while their technique tackles the problem of branch prediction for C and Fortran programs, our work investigates the frequency estimation of virtual call targets which are specific to object-oriented programs. Furthermore, the static features used in their approach are mostly based on characteristics of instructions and control flows. In contrast, Festival focuses on features at higher levels, since it tries to reveal design intentions. Buse and Weimer [10] recently introduced a machine learning-based approach to estimation of execution frequency for program paths. Inspired by this work, we choose our static features to capture design intentions. Different from Festival, their approach is focused on program paths rather than virtual calls. Moreover, their main idea is to perform estimation based on state change patterns. It is different from our idea, which is mainly about the specialty and popularity of target methods and classes.

Dynamic profile-guided techniques are widely used to predict program behaviors to support code optimizations. Grove et al. [16] developed the call chain profile model to describe profile information at various granularities. In their work, they have performed a detailed study on the predictability of receiver class distributions which shows that the distributions are strongly peaked and stable across both inputs and program versions. Although taking a different prediction approach, our work is largely motivated by the results of the study. Virtual method calls and switch statements are usually implemented by indirect jumps at the instruction level. Li and John [21] explored the control flow transfer behaviors of Java runtime systems. Besides other observations, they found that most of the dynamic indirect branches are multi-target virtual calls and a few target addresses have very high frequencies, which confirms the results of the study by Grove et al. Other dynamic techniques [14] [19] have been proposed to improve the prediction accuracy and reduce the misprediction penalty for indirect branches. Compared with these dynamic techniques, Festival is relatively lightweight in that it only requires surface level instrumentation and program-based features. No instruction level manipulation or hardware extension is needed. In addition, the estimation phase of Festival is purely static and does not rely on representative workloads.

Besides dynamic profiling, static techniques have also been proposed to estimate frequencies of various program elements. Wall [23] conducted a comprehensive study on how well real (dynamic) and estimated (static) profiles can predict program behaviors. Based on the study, Wall argued that real profiles are usually better than estimated profiles. However, he also warned about the representativeness of real profiles. Focusing on non-loop two-way branches, Ball and Larus [6] proposed several heuristics to perform program-based branch prediction for programs written in C and Fortran. Based on these heuristics, Wu and Larus [25] designed a group of algorithms to statically calculate the relative frequencies of program elements. They use Dempster-Shafer technique to combine basic heuristics into stronger predictors. To address the similar issue, Wagner et al. [22] independently developed a static estimation technique. They used Markov model to perform inter-procedural estimations. Similar to our

approach, these static approaches aim at the problems which are not amenable to dynamic techniques. Therefore, their motivations also greatly motivate our work. However, we focus on virtual calls in object-oriented programs that have not been studied by the existing work.

7 Conclusions and Future Work

In this paper we have described the Festival approach to frequency estimation for virtual call targets in object-oriented programs. Using static feature data and dynamic profiles of selected programs, we train a multilayer perceptron model and use it to perform estimation for new programs. The evaluation shows that Festival can provide estimations which are much more accurate than estimations based on the uniform frequency distribution. It means that the approach can be useful to a number of applications.

In our future work, we are planning to investigate more static features from other aspects (e.g., control-flow graph structure) and figure out how to combine probabilistic points-to analysis with Festival in order to make them benefit from each other.

Acknowledgments. We are grateful to Qingzhou Luo for his discussion and to Eric Bodden for his suggestions on using Soot and TamiFlex. The presented work was supported in part by National Natural Science Foundation of China (NSFC) (Grants No.60673120 and No. 60970009).

References

1. ANTLR Parser Generator, <http://www.antlr.org/>
2. CLOC – Count Lines of Code, <http://cloc.sourceforge.net/>
3. Kendall tau distance, http://en.wikipedia.org/wiki/Kendall_tau_distance
4. Soot: a Java Optimization Framework, <http://www.sable.mcgill.ca/soot/>
5. Baah, G.K., Podgurski, A., Harrold, M.J.: The probabilistic program dependence graph and its application to fault diagnosis. In: ISSSTA 2008: Proceedings of the 2008 International Symposium on Software Testing and Analysis, pp. 189–200. ACM, New York (2008)
6. Ball, T., Larus, J.R.: Branch prediction for free. In: PLDI 1993: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation, pp. 300–313. ACM, New York (1993)
7. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, pp. 169–190. ACM, New York (2006)
8. Bodden, E., Sewe, A., Sinschek, J., Mezini, M.: Taming Reflection (Extended version). Technical Report TUD-CS-2010-0066, CASED (March 2010), <http://cased.de/>

9. Boogerd, C., Moonen, L.: Prioritizing software inspection results using static profiling. In: Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 149–160 (2006)
10. Buse, R.P.L., Weimer, W.: The road not taken: Estimating path execution frequency statically. In: ICSE 2009: Proceedings of the 31st International Conference on Software Engineering, pp. 144–154. IEEE Computer Society, Washington, DC, USA (2009)
11. Calder, B., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M., Zorn, B.: Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.* 19(1), 188–222 (1997)
12. Chambers, C., Dean, J., Grove, D.: Whole-program optimization of object-oriented languages. Technical report, Department of Computer Science and Engineering, University of Washington (1996)
13. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
14. Ertl, M.A., Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters. In: PLDI 2003: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation, pp. 278–288. ACM, New York (2003)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
16. Grove, D., Dean, J., Garrett, C., Chambers, C.: Profile-guided receiver class prediction. *SIGPLAN Not* 30(10), 108–123 (1995)
17. Hovemeyer, D., Pugh, W.: Finding bugs is easy. *SIGPLAN Not* 39(12), 92–106 (2004)
18. Hwang, Y.-S., Chen, P.-S., Lee, J.K., Ju, R.D.-C.: Probabilistic points-to analysis. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 290–305. Springer, Heidelberg (2004)
19. Joao, J.A., Mutlu, O., Kim, H., Agarwal, R., Patt, Y.N.: Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In: ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 80–90. ACM, New York (2008)
20. Lhoták, O.: Spark: A flexible points-to analysis framework for Java. Master's Thesis, McGill University (2002)
21. Li, T., John, L.K.: Understanding control flow transfer and its predictability in Java processing. In: IEEE International Symposium on Performance Analysis of Systems and Software, pp. 65–76 (2001)
22. Wagner, T.A., Maverick, V., Graham, S.L., Harrison, M.A.: Accurate static estimators for program optimization. In: PLDI 1994: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, pp. 85–96. ACM, New York (1994)
23. Wall, D.W.: Predicting program behavior using real or estimated profiles. *SIGPLAN Not* 26(6), 59–70 (1991)
24. Witten, I.H., Frank, E.: *Data Mining: Practical Machine Learning Tools and Techniques*, 2nd edn. Morgan Kaufmann, San Francisco (2005)
25. Wu, Y., Larus, J.R.: Static branch frequency and program profile analysis. In: MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 1–11. ACM, New York (1994)

Counting Messages as a Proxy for Average Execution Time in Pharo

Alexandre Bergel

PLEIAD Lab, Department of Computer Science (DCC),
University of Chile, Santiago, Chile

<http://bergel.eu>, <http://pleiad.dcc.uchile.cl>

Abstract. Code profilers are used to identify execution bottlenecks and understand the cause of a slowdown. Execution sampling is a monitoring technique commonly employed by code profilers because of its low impact on execution. Regularly sampling the execution of an application estimates the amount of time the interpreter, hardware or software, spent in each method execution time. Nevertheless, this execution time estimation is highly sensitive to the execution environment, making it non reproductive, non-deterministic and not comparable across platforms.

On our platform, we have observed that the number of messages sent per second remains within tight ($\pm 7\%$) bounds across a basket of 16 applications. Using principally the Pharo platform for experimentation, we show that such a proxy is stable, reproducible over multiple executions, profiles are comparable, even when obtained in different execution contexts. We have produced COMPTEUR, a new code profiler that does not suffer from execution sampling limitations and have used it to extend the SUnit testing framework for execution comparison.

1 Introduction

Software execution profiling is an important activity to identify execution bottlenecks. Most programming environments come with one or more powerful code execution profilers.

Profiling the execution of a program is delicate and difficult. The main reason is that introspecting the execution has a cost, itself hardly predictable. This situation is commonly referred to the Heisenberg effect¹. Profiling an application is essentially a compromise between the accuracy of the obtained result and the perturbation generated by the introspection.

Execution profiling is commonly achieved via several mechanisms, often complementary: simulation [26], application instrumentation, and periodically sampling the execution, typically the method call stack. Sampling the execution is favored by many code profilers since it has a low overhead and it is accurate for a long application execution. Execution sampling assume that the number of

¹ “Observation that the very act of becoming a player changes the game being played.”, <http://www.businessdictionary.com/definition/Heisenberg-effect.html>

samples for a method is proportional to the time spent in the method. Profilers uses execution sampling to estimate the amount of time an interpreter, the CPU or a virtual machine, has spent in each method of the program.

Nevertheless, execution sampling is highly sensitive to garbage collection, thread scheduling and characteristics of the virtual machine, making it non-deterministic (e.g., the same execution, profiled twice, does not generally give two identical profiles) and tied to the execution platform (e.g., two profiles of the same execution realized on two different virtual machines or operating systems cannot be meaningfully related to each other). As a consequence, the method execution time estimate is highly variable across multiple executions and closely dependent on the execution environment.

Pharo² is an emerging object-oriented programming languages that is very close to Smalltalk, is syntactically simple, has a minimal core and with few but strong principles. In Pharo, sending a message (also termed “invoking a method” or “calling a method”) is the primitive syntactic construction from which all computations are expressed. Class and method creation, loops, and conditional branches are all realized via sending messages. As coined by Ungar *et al.* when referring to Smalltalk, “the pure object-orientation of the language implies a huge number of messages which are often time-consuming in conventional implementations [23]”. The results presented in this paper were obtained with Pharo.

This paper argues that counting message sends has strong benefits over estimating the method execution time from execution sampling in Pharo. Since Pharo realizes a computation almost exclusively by sending messages, it is natural to evaluate whether counting messages can be used as a proxy for estimating the application execution time.

The three research questions addressed in this paper are:

- A - *Is the number of sent messages related to the average execution time over multiple executions?*
- B - *Is the number of sent messages more stable than the execution time over multiple executions?*
- C - *Is the number of sent messages as useful as the execution time to identify an execution bottleneck?*

This paper answers these three questions positively after careful and extended measurements in different execution settings. We show that counting the number of sent messages is an accurate proxy for estimating the execution time of an application and of an individual method.

Naturally, the execution time of a piece of code is not solely related to the number of invoked methods. Garbage collection, use of primitives offered by the virtual machine, and native calls are likely to contribute to the execution time. However, for all the applications we have considered in our experiments, these factors represent a minor perturbation. The number of method invocations is highly correlated with the average execution time for 10 successive executions (correlation of 0.99 when considering the application execution and 0.97 when

² <http://www.pharo-project.org>

considering individual methods). Moreover, counting messages is more stable over multiple executions, with a variability ranging from 0.06% to 2.47%. The execution time estimated from execution sampling has a variability ranging up to 46.99% (!). For our application setting, we show that measuring the number of sent messages is about 22 times more stable than the measured execution time. The main innovations and contributions of this paper are as follows:

- the limitations of execution sampling are identified (Section 2)
- for a number of selected applications, we show empirically that the number of message sends is a more stable criterion for profiling than execution sampling for each application (Section 3) and individual method (Section 4)
- we describe a general model for evaluating the stability and precision of profiles over multiple executions (Section 4.4)
- we propose an extension of the xUnit framework to compare execution based on the Compteur profiler (Section 5)

Subsequently, key implementation points are presented (Section 6). Reflections and lessons learnt are given next (Section 7). We then review the related work (Section 8) before concluding (Section 9).

2 Profiling Based on Execution Sampling

Profiling is the recording and analysis of which pieces of code are run, and how frequently, during a program's execution. Profiling is often considered essential when one wants to understand the dynamics of a program's execution. A profiler has to be carefully designed to provide a satisfactory balance between accuracy and overhead.

However execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the collection of methods being executed. Such a profiling technique has little impact on the overall execution. Almost all mainstream profilers (JProfiler³, YourKit⁴, xprof^[12], hprof⁵) use execution sampling. Execution sampling comes with a number of serious issues. As we will see, some of these issues have already been pointed out by other researchers. Nevertheless we have chosen to list them in this section for the sake of completeness, and because we will address them in the forthcoming sections.

This section is presented from the point of view of the Pharo programming language.

Dependency on the executing environment. Execution sampling is highly sensitive to the executing environment. As one may expect, running other threads or OS processes while profiling is likely to consume resources including CPU and memory which could invalidate the measurements. Most operating systems

³ <http://www.ej-technologies.com>

⁴ <http://www.yourkit.com>

⁵ <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>

use the *multilevel feedback queue* algorithm to schedule threads [15]. The algorithm determines the nature of a process and gives preferences to short and input/output processes. The thread scheduling disciplines offered by operating systems and/or virtual machines makes thread scheduling a source of measurement perturbation that cannot reliably be predicted. One of the reasons is that no enforcement is made to consistently execute a task in a delimited amount of time: writing a simple email makes concurrently executing programs execute a few CPU cycles longer.

Execution sampling traditionally requires virtual machine support⁶ or an advanced reflective mechanism. In Pharo, execution sampling is realized via a thread running at a high priority that regularly introspects the method call stack of the thread that is running the application. Scheduling new threads, or varying the activity of existing threads (e.g., a refresh made by the user interface thread), is a source of perturbation when measuring execution time since a smaller share of the total profiled execution time is granted to the thread of interest.

Garbage collection is another significant source of perturbation since the profiled application process shares the memory and the garbage with other processes. A memory scan (necessary when scavenging unused objects) suspends the computation, but adds to the application execution time. Garbage collection occurs when memory is in short supply and is hence not exactly correlated with any particular execution sequence.

These problems are not Pharo-specific. They are found in several common execution platforms, as mentioned by Mytkowicz *et al.* [21,22]. There are numerous other sources of measurement bias, for example the relation between the sampling period and the period of thread scheduling [21]. Randomly collecting sampling has been proved to be effective in reducing some of the problems related to execution sampling [21], however, it does not address the non-determinism and the lack of portability.

Non-determinism. Regularly sampling the execution of an application is so sensitive to the executing environment that it makes the profiling non-deterministic. Profiling the very same piece of code twice does not produce exactly the same profile. Consider the Pharo expression `30000 factorial`. On an Apple MacBook Pro 2.26Ghz, evaluating this expression takes between 3 803 and 3 869 ms (ranges obtained after 10 executions). The difference may be partially explained due to the variation of the garbage collection activity. Computing the factorial of 30 000 triggers between 800 and 1000 incremental garbage collections in Pharo. The point we are making is not that the implementation of the factorial function requires a garbage collector, but that a single piece of code may induce significant variation in memory activity.

A common way to reduce the proportion of random perturbations is to ensure that the code to be profiled takes a long execution time. By doing so, the effect of the garbage collector is minimized. Long profiling periods are relatively accurate,

⁶ e.g., <http://download.oracle.com/javase/1.4.2/docs/guide/jvmpi/jvmpi.html>

however, it makes code profiling an activity that may not be practiced as often as a programmer would like.

Lack of portability. Profiles based on execution sampling are not reusable across different runtime execution platforms [6], virtual machines and CPUs. A profile realized on a platform *A* cannot be easily related to a similar profile realized on a platform *B*. For example, the first version of the Mondrian visualization engine [18] was released in 2005 for Visualworks Smalltalk⁷. In 2008 Mondrian development was moved to Pharo. Since its beginning Mondrian has been constantly profiled to meet scalability and performance requirements. However, because of (i) the language change from Visualworks to Pharo, (ii) the constant evolution of Pharo and (iii) the continuous evolution of the physical machine and the Pharo virtual machine, profiles cannot meaningfully be related to each other.

Shared resources. In addition to the general issues mentioned above, a particular profiler implementation comes with its own limitations.

Memory is a persistent global shared resource. Executions that were completed before beginning the profiling may leave the memory in such a state that the application is prone to excessive garbage collection. In Pharo, the programming environment uses the same memory heap that is used to run applications. Previous programming activity may therefore impact it.

MessageTally, the standard profiler of Pharo, constructs a profile sharing the same memory space as the running application, which is a favorable condition for the Heisenberg effect. The longer the application execution takes, the more objects are created by MessageTally to model the call graph and store runtime information, thus exercising additional pressure on the memory manager.

3 Counting Messages as a Proxy for Execution Time

Almost all computation in Smalltalk, and thus in Pharo, is realized via sending messages. Operations like conditional branching and arithmetic are essentially realized via sending messages.

In such an environment, it seems possible that CPU time is likely to be related to the number of messages sent.

3.1 Execution Time and Number of Message Sends

Determining whether the number of messages sent during the execution of an expression is related to the time taken for the expression to execute is a bit trickier than it appears. Execution time measurements are hardly predictable. As with any statistical measurement, the correlation between two variables is realized by bounding the error margin in the measurement. The relation is established if this margin is “small enough”. Determining a relation between two data sets requires a number of statistical tools [16]. We will follow the traditional steps of constructing a regression model.

⁷ <http://www.cincomsmalltalk.com/main/products/visualworks/>

Intuitively, we expect the number of messages sent during the execution of an expression to increase with an increase of the execution time: the longer an expression takes to execute, the more messages are sent. We will later discuss native calls and other interactions with the operating system. This subsection answers research question A.

Measurements. From the Pharo ecosystem⁸ we selected 16 Pharo applications. We selected these applications based on their coverage of Pharo. Appendix A lists the applications and gives the rationale for choosing them. The experiment was conducted on a MacBook Pro 2.26 GHz Intel Core 2 Duo with OSX 10.6.4 and 2GB 1067 MHz DDR3 using the SqueakVM Host 64/32 Version 5.7b3 (this execution context is designated as *c* in the following sections).

Our measurements, used to relate the number of sent messages the execution time, have to be based on representative application executions, close to what programmers are experiencing. Running unit tests is convenient in our setting since unit tests are likely to represent common usage and execution scenarios [17]. We execute the unit tests associated with each of the 16 applications. None of the tests we used in this paper manipulates randomly generated data or makes use of non-deterministic data input. The execution time and the number of message sends are measured for each test suite execution. As an illustration of the message-send metric we are interested in, consider the following code (which is a simplified version of a test from Moose, a platform for software analysis):

```
ModelTest>> testRootModel
  self assert: MooseModel new mooseID > 0
Behavior>> new
  ^ self basicNew initialize
Behavior>> basicNew
  <primitive: 70>
Object>> initialize
  ^ self
MooseElement>> mooseID
  ^ mooseID
```

The test `testRootModel` sends 6 messages. The messages `assert:`, `new`, `mooseID` and `>` are directly sent by `testRootModel`. The message `new` sends `basicNew` and `initialize`. The total number of messages sent by `testRootModel` is 115. The message `assert:`, which belongs to the SUnit framework, does some checks on the argument and the method `initialize` is redefined in the class `MooseElement`.

The number of messages can easily skyrocket. Running the tests associated with the Pharo collection library [10] takes slightly more than 32 seconds. The test execution sends more than 334 million messages.

Linear regression. A scatter plot is drawn from our measurements (Figure 11). Each of the applications we have profiled is represented by a point (*execution time, number of message sends*) and is denoted with a cross in the scatter plot. The measurements *execution time* and *number of message sends* are the average

⁸ Principally available from <http://www.squeaksource.com>

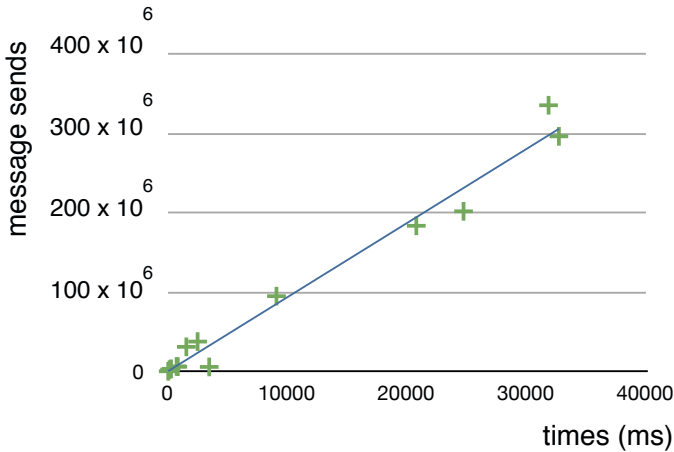


Fig. 1. Linear regression for the 16 Pharo applications

of 10 successive executions. These values form an almost straight line with a statistical correlation of 0.99. The correlation is a general statistical relationship between two *random variables* and *observed data values*: a value of 1 means the data forms a perfect straight line. This line, commonly called regression line, may be deduced from these values.

The general equation of a regression line is $\hat{y} = a + bx$ where a is constant term; b is the line slope; x is the independent variable; y is the dependent variable; \hat{y} the predicted value of y for a given value of x . The *independent variable* is the execution time and the *dependent variable* is the number of message sends. We also put an additional constraint on the constant term: an execution time of 0 means that no message has been sent.

Using the material provided in Appendix A, we estimate the sample regression line on our machine to be $\hat{y} = 9\,335.55x$, meaning that in the average, the virtual machine sends 9.3 million of messages per second. The line is drawn in Figure 1.

We designate the average message rate (number of message sends per unit of time) as $MR_{\Gamma,c}$ where Γ is the set of the applications we profile and c the context in which the experiment has been realized. c captures all the variables that the measurements depends on (e.g., computer, RAM, method cache implementation, temperature of the room).

We now have established the relation between the number of message sends and the execution time. We are not done yet however: only an approximation has been determined. The $MR_{\Gamma,c}$ value has been computed from an arbitrary set of applications. If we had chosen a different set of applications, say Γ' , $MR_{\Gamma',c}$ would have probably be slightly different from $MR_{\Gamma,c}$. $MR_{\Gamma,c}$ is said to be a random variable, and it possesses a probability distribution. Assuming that the applications we have chosen are representative of the all possible applications available in Pharo, the real value of $MR_{\mathbb{A},c}$, where \mathbb{A} is the set of all Pharo applications, rests in an interval that is calculated according to how confident we want to be in our findings.

The standard deviation of error tells us how widely the errors are spread around the regression line. This value is essential to estimate the confidence interval that includes $MR_{A,c}$. Appendix [A](#) details how the standard deviations (s_e and s_b) are computed. We have the standard deviation of error $s_e = 16\,448\,897$. The confidence interval is $[MR_{\Gamma,c} - t s_b; MR_{\Gamma,c} + t s_b]$ where $s_b = 350.84$ is the standard deviation of $MR_{\Gamma,c}$ and t is a value obtained from the standard t distribution table based on the confidence $(1 - \alpha)$ we want to have, with 1 (= 100%) being the most confident.

For a 95% confidence interval, we have $\alpha = 0.05$ and therefore $t = 2.145$ according to the standard t distribution, which may be found in any statistical text book. As a result, the confidence interval is $[8\,582, 10\,089]$, which means that there is a probability of 95% that the real value $MR_{A,c}$ is within the interval.

The linear regression model enables the prediction of the average execution time from the number of sent messages. Consider GitFS, an implementation of Git in Pharo. The tests of GitFS send 28 096 569 messages. According to the regression model, this corresponds to a period of time $(28\,096\,569 - 418\,253)/9\,335.55 = 2\,965$. GitFS' tests actually run in 2 928 milliseconds, which is included in the time interval $[2\,743, 3\,225]$.

3.2 Method Invocation

One of the problems message counting is addressing is the poor stability and prediction of execution sampling. This section compares the stability of the execution time with the stability of number of sent messages, which answers the research question *B*.

Hash values. Before we further elaborate on the precision of message counting, it is relevant to remark that executing the *same* code expression multiple times may not always send the same number of messages. For example, adding an element to a *set* does not always send the same number of messages. Consider the following code excerpt:

```
| s |
s := Set new.
Compteur numberOfCallsIn: [ 1000 timesRepeat: [ s add: Object new ] ]
```

Line 2 creates a new set. Line 3 invokes our library by sending the message `numberOfCallsIn:` which takes a block as parameter (a block is equivalent to a lambda expression in Scheme and Lisp and an anonymous inner class in Java). Line 4 creates 1000 entries in the set. The hash values of the key objects are used for the internal indexing of the set. The virtual machine generates the hash values and they cannot be predicted since they are based on a pseudo random number generator⁹. Each execution of this piece of code gives a different value (e.g., 54 383, 55 997, 56 165) since the computation needed to add an object

⁹ In Cog, the jitted virtual machine, the hash is derived from the memory allocation pointer. The non-jitted VM produces a new hash value from a formula taking as input the previous generated hash value.

into a table depends on the object hash value pseudo-randomly provided by the virtual machine.

Even though the way hash values are assigned to objects is indeed a source of non-determinism, as we will subsequently see, it has a low impact on our measurement: for the applications we have profiled, the number of message invocations varies significantly less than the execution time. Interactively acquiring data from the user, the filesystem or the network may also be another source of variation for the number of message sends.

Coefficient of variation. Each execution of the same piece of code results in a different execution time and a different number of messages sent. We will now assess whether the number of sent messages is a more stable metric than the execution time over multiple executions. For each of the 16 applications we executed its tests 10 times and calculated the standard deviation of execution time ($s_{TimeTaken}$) and number of sent messages ($s_{messages}$). To be able to compare these two standard deviations, we use the coefficient of variation, defined as the ratio of the standard deviation to the mean, resulting in c_{time} and $c_{messages}$, respectively. Appendix A gives our measurement and details how the variation is computed.

For the 16 applications we considered, our result shows that the stability of the execution time (the c_{time} column) varies significantly from one application to another. For example, the applications ProfStef, Glamour and Magritte are relatively constant in their execution time. The variation may even be below 1% for ProfStef. However, execution time significantly changes at each run for a number of the applications. The execution time of XMLParser, DSM and PetitParser varies from 25% to 46%. The execution time of PetitParser may vary by 46% from one run to another. The reason for this is not completely clear. Private discussion with the author of PetitParser revealed the cause of this variation to be the intensive use of short methods on streams. These short methods, such as `peek` to fetch one character from a stream and `next` to move the stream position by one, have an execution time close to the elementary operations performed by the virtual machine to lookup the message in method cache¹⁰.

In contrast to execution time, message counting is a much more stable metric since its variation is usually below 1%. The greatest variation we have measured are with Mondrian and Moose. This is not surprising since these two applications intensively use non-deterministic data structures like sets and dictionaries to store their model.

The average values of the normalized standard deviation of execution time c_{time} and $c_{messages}$ are 13.95 and 0.61, respectively. For the experimental set up we have used, we have found that over multiple executions of the same piece of code, measuring the number of sent messages is 22.86 (13.95 / 0.61) times more stable than measuring the execution time.

¹⁰ The reason why fast messages cause the execution time to vary so much is not completely clear to us. We cannot reproduce this on micro-benchmarks. Additional analyses are required. We have left this as future work.

3.3 Effect of the Execution Context

We repeated the experiment on two additional execution platforms: on the MacBook Pro using the Cog virtual machine (which supports Just-In-Time compilation (JIT)) and a Linux Gentoo (2.6.34-gentoo-r6 running on an Intel Xeon CPU 3.06GHz GenuineIntel) using a non-jitted virtual machine.

On the Cog virtual machine we have $MR_{\Gamma, c'} = 58\,384.75$, with a 95% confidence interval [55 325, 64 123]. On this platform, the ratio between c_{time} and $c_{messages}$ is 18.98. This is lower than what we obtained on the non-jitted virtual machine. The reason stems from the multiple method compilations, each being a resource-consuming process on its own.

On the standard virtual machine running on Linux we obtained $MR_{\Gamma, c''} = 12\,412.34$, with a 95% confidence interval of [9 615, 14 121]. The ratio between c_{time} and $c_{messages}$ is 22.34, which corresponds to the ratio we have measured on the MacBook Pro without the JIT-ing VM.

3.4 Tracking Optimizations

We identified a number of execution bottlenecks in the Mondrian visualization engine in our previous work [5]. We removed the bottlenecks by adding a “memoization” mechanism which is a common technique applied to methods free of side effects to avoid unnecessary recalculations. Memoizing the method `MOGraphElement>> bounds` improved Mondrian performance by 43%. Another memoization of `MOGraphElement>> absoluteBounds` resulted in a speedup of 45% (for the UI thread this time). Comparing the number of message sends with and without the optimization gives performance increases in the same range: the number of messages sent with the `bounds` optimization is 42% less than the non-optimized version and 44% for the `absoluteBounds` optimization.

We have sequentially measured the logic thread then the UI thread. After having sampled the execution of the logic thread we had to restart the virtual machine to use the same initial state of the memory and the method cache. There was no need to restart the virtual machine between the two measurements when counting messages. No major conclusion can be drawn from this experiment. However, it emphasizes an important practical point.

3.5 Cost of Counting Messages

Counting the number of executed send bytecode instructions is cheap. We measure the execution time of each of the 16 applications with and without the presence of message counting. Table 3 reports our results. Each measurement is the average of 5 executions. The overhead is computed as $overhead = (time\ on\ modified\ VM - time\ on\ normal\ VM) / time\ on\ normal\ VM * 100$.

The cost of message counting is almost insignificant. The execution time variation ranges from 0% to 0.02%. These results are not surprising actually. Message counting is simple to implement within the virtual machine; at each send bytecode a global variable is incremented. This is a cheap operation compared to the

complex machinery to lookup method implementation, interpret the bytecode, and to manage the memory. The execution time variation we have measured on a non-jitted virtual machine is of the same range on Cog.

4 Counting Messages to Identify Execution Bottlenecks

CPU profilers aim at identifying methods that consume a large share of the execution time. These methods are likely to be considered for improvement and optimization, aiming at reducing the total program execution time. This section considers counting message as a means of finding runtime bottlenecks, and answers research question *C*.

4.1 A Method as an Execution Bottleneck

A method is commonly referred as an execution bottleneck when it is perceived as taking a “lot of time”, or more time that it should. The intuition on which we will elaborate is that if a method is slow then it is likely to be sending (directly and indirectly) “too many” messages.

Sending “too many” messages may not be the only source of slow down. An excessive use of memory and numerous invocations of the primitives offered by the virtual machine are likely to play a role in the time taken for a program to execute. A program that intensively uses files or the network may spend a significant amount of time executing the corresponding primitives. In Pharo, executing a primitive suspends the program execution and resumes it once the primitive has completed. Consider a program that sends few messages but makes a great use of primitives: the program can take a long time to execute with few sent messages. However, we have not detected such occurrence in all the applications we studied. As we will see in the coming sections, in spite of the perturbation that may be introduced by primitive executions, still make message counting more advantageous than execution sampling for all the applications we have considered.

4.2 Method Invocations Per Method

Counting the number of messages sent by a particular method is an essential step to compare execution sampling with message counting.

Counting the number of sent messages for each method requires associating with each method the number of messages it sends at each execution. Most code instrumentation libraries and tools, including most aspect-oriented programming ones, easily meet this requirement. The instrumentation we consider for each method of the application to be profiled is done as follows.

```
CompteurMethod>> run: methodName with: listOfArguments in: receiver
| oldNumberOfCalls v |
oldNumberOfCalls := self getNumberOfCalls.
v := originalMethod valueWithReceiver: receiver arguments: listOfArguments.
numberOfCalls := (self getNumberOfCalls - oldNumberOfCalls) + numberOfCalls - 5.
^ v
```

Compteur is the implementation of our message-based code profiler for Pharo. An instance of the class *CompteurMethod* is associated with each method of the application to be profiled. *CompteurMethod* acts as a method wrapper by intercepting each method invocation. At each method invocation, the method `run:with:in:` is executed to increase the variable `numberOfEmittedCalls` defined in the *CompteurMethod* instance. The number of executions of a method is associated with the method itself. Note that we do not instrument the whole system, but just the application we are interested in profiling. The method `getNumberOfCalls` uses a primitive operation defined in the virtual machine to obtain the current number of message sends.

The instrumentation itself sends 5 messages: `valueWithReceiver:arguments:`, `withArgs:executeMethod:` and the second `getNumberOfCalls`, plus 2 messages sent by `valueWithReceiver:arguments:`, not presented here. We therefore need to subtract 5 from the number of calls.

4.3 Method Execution Time and Number of Message Sends

The total execution time of an individual method is correlated with the number of messages that are directly and indirectly sent by the method. In this section, we focus on a single application, Mondrian. Other applications enjoy the correlation.

Figure 2 plots the methods of the Mondrian application according to their execution time in milliseconds with the number of sent messages. Note that we consider the total execution time and the total number of message sends for each method, counting the closure of all of the methods that it invokes. This means that if a method is invoked 100 times for which each execution takes 2 ms and sends 5 messages, then the method is plotted as the point (200, 500). The graph shows that the time taken by the computation that is initiated by

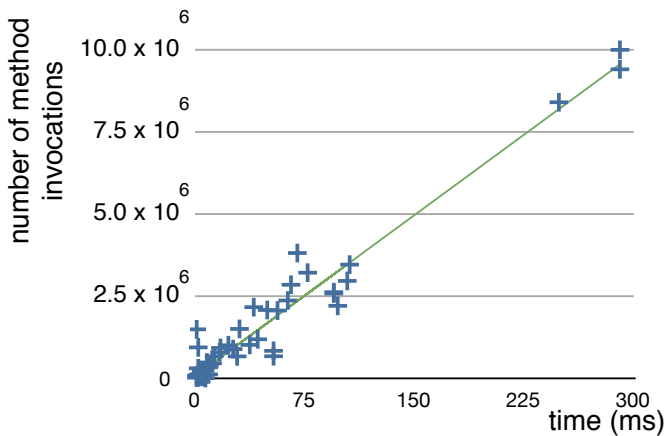


Fig. 2. Linear regression for the methods of Mondrian.

sending a message is almost constant: the execution time of a method is directly proportional to the number of messages that it sends.

As with application execution (Section 3.1), the regression model indicates that the large majority of (*execution time*, *number of message sends*) plots form a straight line (Figure 2), confirmed by a correlation of 0.97.

The equation of the regression line is $\hat{y} = 31\,811.38 x$. Figure 2 gives this line. We see that the slope of the regression line is about 3.4 greater than the slope we found when we studied application executions (Section 3.1). The reason stems from the cumulative effect of nested message sends. To get a feeling why this happens, consider the following two methods:

```
MOGraphElement>> bounds
| basicBounds |
boundsCache ifNotNil: [ ^ boundsCache ].
self shapeBoundsAt: self shape ifPresent: [ :b | ^ boundsCache := b ].
...

MONode>> startPoint
^ self bounds bottomCenter
```

The method `bounds` sends 239 direct and indirect messages. The method `startPoint` sends 2 direct messages. But since it invokes `bounds` and `bottomCenter` (which sends 27 messages), in total, `startPoint` sends $2 + 27 + 239 = 268$ messages.

4.4 Stability of Message Counting

To assess the stability of message counting over execution sampling we will compare a list of profiles made with message counting and execution sampling. The idea is to numerically assess the variability of the method ranking against multiple profiles of the same code execution. We will then characterize a stable set of profiles with a constant method ranking.

Stability of profiles. We have profiled Mondrian 20 times: 10 using execution sampling and 10 using message counting. Each profile is obtained by running the unit tests and provides a ranking of the methods. Methods are ranked in order of “computational cost”. To save space, Table 1 gives only an excerpt of our measurements: the first 9 methods (names have been shortened to `m1...m9`) are ranked for 5 profiles. The method ranked first is the one that has the greatest share of the CPU execution time; the method ranked last is the one that has consumed the least CPU. The 5 profiles are obtained with `MessageTally`. As stated earlier (Section 2), due to the high sensitivity of the environment, not all the rankings are the same. Quantifying the variation of the method ranking for a set of profiles is the topic of this section.

For each method, we compute the standard deviation of the ranking (s_{es}) to estimate ranking variability. We have $s_{es}(m) = 0$ if the method `m` is always ranked the same across the profiles. The greater s_{es} is, the greater the variability of the ranking.

Table 1. Ranking of the first 9 methods of Mondrian for 5 profiles (execution sampling)

	m1	m2	m3	m4	m5	m6	m7	m8	m9
Profile 1	1	2	3	4	5	6	7	8	9
Profile 2	1	2	3	4	6	5	10	12	7
Profile 3	1	2	3	4	6	5	10	12	7
Profile 4	1	2	3	4	5	6	9	7	13
Profile 5	1	2	3	5	6	4	9	12	7
Average	1	2	3	4.1	5.4	5.5	8.9	10.4	8.2
Stand. Dev.									
<i>ses</i>	0.000	0.000	0.000	0.316	0.516	0.707	1.197	1.955	1.989

The stability of a set of profiles depends on the variability of the method ranking. However, not all methods deserve to be considered in the same way. We use the discounted cumulated gain [13] to weight the ranking. The point of a weight is that the lower the ranked position of a method, the less valuable it is for the user, because the less likely it is that the user will ever consider the method as being slow. A discounting function is needed which progressively reduces the method score as its position in the ranking increases. We weight a method ranked n as $w(n) = 1/\ln(n+1)$. We define the instability for the first n methods of the set of profiles P as $\psi^n(P) = \sum_{i=1}^n ses(i) * w(n)$, the sum of the weighted standard deviations. According to the excerpt given in Table 1, we have $\psi^9(P) = 0 \frac{1}{\ln(1+1)} + \dots + 0.316 \frac{1}{\ln(4+1)} + 0.516 \frac{1}{\ln(5+1)} + \dots + 1.989 \frac{1}{\ln(9+1)} = 3.177$. A perfectly stable set of profiles P has the value $\psi(P) = 0$.

Experimental setting. We have profiled each application γ 20 times in the execution context c . We have $\gamma \in \Gamma$, where Γ is the list of applications given in Appendix A. 10 of these profiles were obtained using the standard execution sampling. We refer to these 10 profiles as $P_{\gamma,c}$. As mentioned earlier, the execution context in which the applications are profiled is c . The 10 remaining profiles were obtained using message counting, referred as $Q_{\gamma,c}$. We have chosen to consider the same number of methods for each application since not all the applications have the same code size. As previously described, we ran the unit tests to produce the profiles.

Poor stability of execution sampling. The method ranking against the execution time is not constant: each new profile gives a slightly different method ranking. For example, for 8 of the 10 profiles of $P_{PetitParser,c}$, the method ranked 5th in terms of execution time is `PPPredicateTest>> testHex`. However, in the 2 remaining profiles, this method is ranked 35 and 36 (!). After an examination of the tests to make sure they do not randomly generate data, we speculate that this odd ranking is due to a mixture of the problems highlighted at the beginning of this article (Section 2). This kind of variability in the method ranking is hardly avoidable, even though we took great care to garbage collect the memory and release unwanted object references between each profile.

We define $\psi^{10}(P_{\gamma,c})$ over the first 10 methods given by a set of profiles P for an application γ realized in an execution context c . To give a reference point, we artificially build a random data set R on which we can compare ψ of the applications we profile: we randomly generate 10 random rankings. For our random set of profiles, we have $\psi^{10}(R) = 173$ and $\psi^{10}(P_{PetitParser,c}) = 11$. All the remaining ψ^{10} range from 3 to 5.

The greatest instability of the set of profiles we obtained is for PetitParser. PetitParser makes heavy uses of stream and string processing, which perturbs MessageTally, the standard execution sampling profiler of Pharo, for the same reasons mentioned in Section 3.2 (use of short methods).

Perfect stability of message counting. The profiles obtained with message counting have a ψ of 0 for each of the applications we have profiled. This means that the 10 profiles we made for each application do not show a variation in the method ranking according to the number of sent messages. Even though we have seen that the number of method invocations varies slightly (Section 4.2), the data we collected from this experiment show that this does not impact the method ranking. Profiling multiple times always ranks the methods identically.

The stability execution sampling does not equal that of message counting. The stability of message counting is clearly superior to that of execution sampling.

4.5 Cost of the Instrumentation

Determining the number of sent messages for each method requires complete instrumentation of the application to be profiled. This instrumentation introduces an overhead. The cost of the instrumentation depends on the infrastructure used for code transformation. We used the Spy framework [4]. To evaluate our implementation, we performed two set of measurements. For each application, we ran its associated unit tests twice, with and without the instrumentation. Table 4 presents our results.

Running the unit tests while counting message sends for each method has an overhead that ranges from 2% to 2 524%. This overhead includes the time taken to actually instrument and uninstrument the application. When the unit test takes a short time to execute, then the instrumentation may have a high cost. The worst cases are with XMLParser and AST. AST's unit tests take 37 ms to execute. They take 971 ms with the instrumentation, representing an overhead of 2 524%. The AST package is composed of 76 classes and 1 246 methods. XMLParser's unit tests take 36 ms to execute. The package is composed of 47 classes and 785 methods. Since XMLParser is smaller than AST, the overhead of the instrumentation is also smaller.

Figure 3 represents the ratio of the overhead to the test execution time. The left hand-side presents this ratio with a linear scale. The right-hand side gives the same data, with a logarithmic scale for the overhead. Each cross is a pair (*execution time, overhead*), representing an application. Figure 3 shows a general trend: the longer the unit tests take to execute, the smaller the instrumentation overhead. Above an execution time of approximately 5 seconds, determining

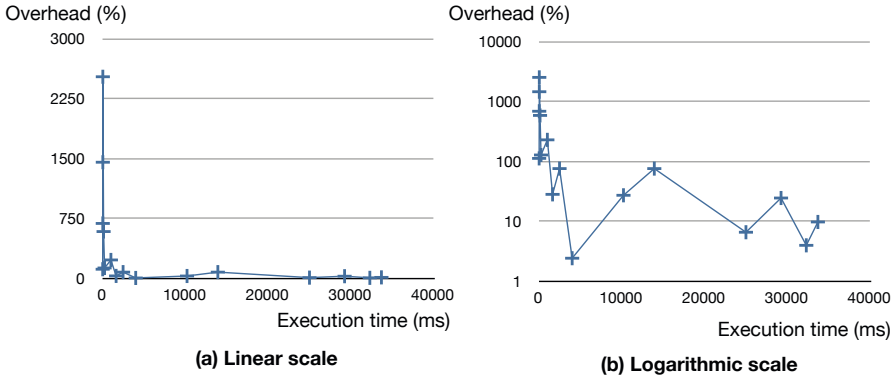


Fig. 3. Ratio between overhead and execution time

the number of message sends per method has an overhead of less than 100%, which represents twice the execution time of the unit tests. In practice, this is acceptable in most of the situations we have experienced.

DSM has an overhead of 2.4%, the smallest overhead we measured. The reason for this low overhead is that most of the logic used by the DSM package is actually implemented in Famix, a different package. When DSM is the only package instrumented, the overhead is low since most of the work happens in a different package, itself uninstrumented.

Note that the execution time of the tests and the cost of the instrumentation are unrelated. This is because the execution time of the tests depends on how much logic is executed to complete tests, and not on how much of that execution is attributable to the execution package.

5 Contrasting Execution Sampling with Message Counting

We revisit the issues encountered with execution sampling that we previously enumerated (Section 2) and contrast them with the message counting technique described above.

No need for sampling. Message counting provides an exact measurement of a particular execution. The measurement is solely obtained by counting the number of message sends. Message counting therefore does not depend on thread support or advanced reflective facilities (e.g., MessageTally heavily relies on threads and runtime call stack introspection) or sophisticated support of the virtual machine (e.g., the JVM offers a large protocol for profiling agents). As described in Section 6, adapting a non-jitted virtual machine to count send instructions may require adding a few dozen lines of code.

Execution environment. Message counting is not influenced by the thread scheduling or memory management. The benefit is that we are able to compare profiles obtained from different execution environments. For the applications we have considered, sending messages is correlated with the average execution time. As we have shown, this means that the average execution time can be easily approximated from the number of messages.

Stable measurements. Measurements obtained from message counting are significantly more stable than those obtained from execution sampling. Even though the exact number of message sends may vary over multiple executions (partly due to the hash values given by the virtual machine), the metric is stable and reproducible in practice.

Profiling time. Contrary to execution sampling, message counting is well adapted to short profiles since an exact value is always returned. One compelling application of this property is asserting upper bounds on message counts when writing tests. We have produced an extension of unit test that offers a new kind of assertion: `assertIsFasterThan:` to compare the number of messages sent.

We have written a number of tests that define time execution invariant. One example for Mondrian is (the difference between the two executions is shown in **bold**):

```
MondrianSpeedTest>> testLayout2
| view1 view2 |
"All the subclasses of Collection"
view1 := MOViewRenderer new.
view1 nodes: (Collection allSubclasses).
view1 edgesFrom: #superclass.
view1 treeLayout.

"Collection and all its subclasses"
view2 := MOViewRenderer new.
view2 nodes: (Collection withAllSubclasses).
view2 edgesFrom: #superclass.
view2 treeLayout.

self assertIs: [ view1 root applyLayout ] fasterThan: [ view2 root applyLayout ]
```

The code above says that computing the layout of a tree of n nodes is faster than with $n + 1$ nodes. The difference between these two expressions is just the message sent to `Collection`. Being able to write a test for short execution time is a nice application of message counting. As far as we are aware, none of the mainstream testing frameworks is able to define assertions to compare execution times.

6 Implementation

COMPTEUR is an implementation of the message-counting mechanism for Pharo. It comprises a new virtual machine and a profiler based on the Spy profiling framework [4].

The modification made in the virtual machine is lightweight: a global variable initialized to 0 is incremented each time a send bytecode is interpreted. In the non-jitted Pharo virtual machine, the increment is realized in the part of the bytecode dispatch switch dedicated to interpret message sending. In the jitted Cog virtual machine, the preamble of the method translated in machine code by the JIT compiler realizes the increment.

The maximum value of a small integer in Pharo is 2^{30} ($\sim 1.073 * 10^9$). Over this value, an integer is represented as an instance of the `LargeInteger` class, which is slow to manipulate within the virtual machine. The current Pharo virtual machine (5.7beta3) executes approximately 12 M message sends per second on micro benchmarks¹¹. This means that the range of the Pharo integer values may be exhausted after 90 seconds ($1\ 073 / 12$).

Using a 64 bit integer is not an option since Pharo is designed to run on 32 bit machines. We therefore use two small integers to encode the number of sent messages. The maximum number of messages that can be counted in this way is $\sim 1.152 * 10^{18}$. Even at full interpretation speed, this value is not reached after 2 million hours.

The global message counter is made accessible within our profiler written in Pharo via primitives. The counter is reset via a dedicated primitive.

The instrumentation is realized by wrapping methods to intercept incoming messages [4].

To obtain the number of message sends per method, the application has to be instrumented to capture the value of the global counter before and after executing the method, as illustrated in Section 4.2. Using the Aspect-Oriented-Programming terminology, such instrumentation is easily realized with *around* or *before* and *after* advice.

7 Discussion

The design of our approach is the result of a careful consideration of various points.

Modifying the virtual machine. Even though the modification we made in the virtual machine is relatively lightweight, we are not particularly enthusiastic about producing a new virtual machine since the Pharo community is not particularly keen on changing the virtual machine. People are often reluctant to use non-standard tools, even if the benefits are strong and apparent.

We have not found a satisfactory alternative. As an initial attempt, before we implemented the work presented in this paper, we made a profiler that counted only the messages sent by the application, and not by dependent libraries and the runtime. The application was instrumented by code modification and the virtual machine was left unmodified. We discovered that the information we gathered was insufficient to demonstrate the properties presented in this paper. As soon as the execution flow leaves the application, no information is recorded until it returns

¹¹ Result of the standard 0 tinyBenchmarks micro benchmark.

to the application. Since it cannot be accurately predicted how long the execution flow will spend outside the application, we could not establish a correlation between the number of messages sent by the application and execution time.

Instrumentation. Our approach requires instrumentation of the application to be profiled: only the methods defined in the application we wish to improve need to be instrumented.

Instrumenting the complete system has not proven to be particularly useful or possible in our situation: (i) if an execution slowdown is experienced, there is no need to look for its cause outside the application we are actually considering; (ii) instrumenting the whole system has a significant runtime cost; (iii) this easily leads to meta-circularity issues since our profiler shares the runtime with the profiled application. Even if recent advances in instrumentation scoping are adopted [24], this increases the complexity of the implementation without a clear benefit. Efficiently handling metacircularity is necessary to profile the profiler itself. However, since the implementation of COMPTEUR is not particularly complex, we have not felt the need to do so.

Special messages. For optimization purposes, not all messages are sent in Pharo. Depending on the name of the message being sent, the Pharo compiler may decide to transform the message send into a particular sequence of bytecode instructions. Consider the message `ifTrue:iffFalse:` with literal two block arguments. For example, the expression `(1 < 2) ifTrue: ['Everything is okay'] iffFalse: ['Something is wrong']` is translated into the sequence:

```

76 pushConstant: 1
77 pushConstant: 2
B2 send: <
99 jumpFalse: 27
21 pushConstant: 'Everything is okay'
90 jumpTo: 28
20 pushConstant: 'Something is wrong'
87 pop
78 returnSelf

```

Beside `ifTrue:iffFalse:`, there are other 17 control flow instructions treated as “special messages” by the Pharo compiler. In the Pharo virtual machine, a jump bytecode is faster than a send. In the whole Pharo library, approximately 62% of all message sends contained in the source code are translated into send bytecode instructions. The correlation we established between execution time and message sends is strong, even if 38% of message sends are not translated into send bytecode instructions. We obtained these figures by comparing for each method in Pharo the abstract syntax tree of the source code and the generated bytecode instructions.

The case of primitives. The execution time of a method may have little relation to the number of messages sent. This could happen if the method intensively uses primitives, or if the program had to wait for a keystroke. The profile of such a program then depends on how long the user has waited before pressing a key. We reasonably assume that this is not what happens in practice: we chose unit tests as the execution reference, which is a realistic approximation of a program execution.

Particularities of Pharo. Pharo’s compiler is rather simplistic, not designed for producing optimized bytecode. It does not offer additional optimization than a mere bytecode generation pattern based on the name (e.g., `ifTrue:iffalse:`, `timesRepeat:`), as previously mentioned. Pharo memory layout is based on a generational and compacting mark and sweep. Virtual memory file mapping is supported and the virtual machine has the ability to grow and shrink the memory space.

Are our results applicable to other dynamic languages? At first glance, Jython^[12], JRuby^[13], Groovy^[14] enjoy the same nice properties as Pharo: the computation is solely realized via sending messages. It is therefore tempting to extrapolate our results to these languages. However, the Java Virtual Machine, which is the execution platform of these languages, has a radically different execution model. For example, the JVM has native threads which have an impact on the memory management. Pharo has “green threads”: the scheduler is implemented in Pharo itself. Most implementations of the JVM garbage collector have many more generations than the one of Pharo: Pharo supports only 2 generations (young and old) whereas the HotSpot Java VM has 5 memory pools. Last but not least, the heavy optimization of the JVM just-in-time compiler has the potential to completely invalidate our finding. Testing whether our results are applicable to other “pure OO” languages implies further analysis and measurements.

8 Related Work

The work presented in this paper is not the first attempt at finding an alternative to execution sampling. However we are not aware of any work which studied the number of message sends.

Bytecode instruction counting. Comesi *et al.* [6] pioneered the field by investigating the use of bytecode instruction counting as an estimate of real CPU consumption. For all the platforms they have considered, there is an application-specific ratio of bytecode instructions per unit of CPU time. Such a bytecode ratio can be used as a basis for translating a bytecode instruction value into the corresponding CPU consumption.

Our results are similar. We have also identified a message ratio, however this ratio is attached to a particular execution platform, and not to an application.

Dynamic bytecode instrumentation. Instrumentation-based profiling has a high cost. However, such overhead can be reduced by instrumenting only the subset of the application where a bottleneck is known to be. Dmitriev [8] proposes that for a given set of arbitrary “root” methods, instrumentation applies to the call subgraph of the roots only. Dmitriev observed that this approach generally works much better for large applications, than for small benchmarks. The reason is that additional code and data become negligible once the size of the profiled

¹² <http://www.jython.org>

¹³ <http://jruby.codehaus.org>

¹⁴ <http://groovy.codehaus.org>

application goes above a certain threshold. Message counting has similar properties. Only a subset of the system needs to be instrumented. However, message counting behaves perfectly well for small benchmarks.

Hardware Performance Counters. Most modern processors have complex microarchitectures that dynamically schedule instructions. These processors are difficult to understand and model accurately. For that purpose, they provide hardware performance counters [1]. For example, Sun’s UltraSPARC processors count events such as instructions executed, cycles expended and many more.

With message counting we exploit the same kind of information, but obtained from the Pharo virtual machine.

Optimizing Smalltalk. The popularity of Smalltalk during the 80’s has led to numerous works that directly tackled the slow execution of Smalltalk programs. Sophisticated mechanisms on mapping bytecode to instruction machine [23,25], improved compiled methods and cache contexts [19], manipulating method dictionaries [3] and adding type declaration and inference [2,14] have been produced.

9 Conclusion

A code profiler provides high-level snapshots of a program execution. These snapshots are often the only way to identify and understand the cause of a slow execution. Whereas execution sampling is a widely used technique among code profilers to monitor execution at a low cost, it brings its own limitations, including non-determinism and inability to relate profiles obtained from different platforms.

We propose counting method invocations as a more advantageous profiling technique for Pharo. We have shown that having method invocation as the exclusive computational unit in Pharo makes it possible to correlate message sending and average execution time with stability, both for applications as a whole and for individual methods.

We believe that code profiling has not received the attention it deserves: execution sampling uses stack frame identifiers, which essentially ignore the nature of object-oriented programming. In general, code profilers profile object-oriented applications pretty much the same way that they would profile applications written in C. We hope the work presented in this paper will stimulate further research of the field to give more importance to objects than to low-level implementation considerations.

Acknowledgment. We thank Mircea Lungu, Oscar Nierstrasz, Lukas Renggli and Romain Robbes for the multiple discussions we had and their comments on an early draft of the paper. We particular thank Walter Binder for his multiple discussions and advices. Our thanks also go to Eliot Miranda for his help on porting COMPTÉUR to Cog, the jitted virtual machine of Pharo. We thank Gilad Bracha and Jan Vraný for the fruitful discussions we had. We also thank Andrew P. Black for his precious help on improving the paper. We gratefully thank María José Cires for her help on the statistical part. We also thank ESUG, the European Smalltalk User Group, for its financial contribution to the presentation of this paper.

References

1. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. In: Proceedings of PLDI 1997, pp. 85–96. ACM, New York (1997)
2. Robert, G.: Atkinson. Hurricane: An optimizing compiler for Smalltalk. In: Proceedings OOPSLA 1986, pp. 151–158. ACM, New York (1986)
3. Baskiyar, S.: Efficient execution of pure object-oriented programs by follow-up compilation. *Computing* 69, 273–289 (2002)
4. Bergel, A., Bañados, F.B., Robbes, R., Röthlisberger, D.: Spy: A flexible code profiling framework. In: Smalltalks (2010) (to appear)
5. Bergel, A., Robbes, R., Binder, W.: Visualizing dynamic metrics with profiling blueprints. In: Proceedings of TOOLS EUROPE 2010, pp. 291–309. Springer, Heidelberg (2010)
6. Camesi, A., Hulaas, J., Binder, W.: Continuous bytecode instruction counting for cpu consumption estimation. In: Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems, pp. 19–30. IEEE, Los Alamitos (2006)
7. Diwan, A., Lee, H., Grunwald, D.: Energy consumption and garbage collection in low-powered computing. Cu-cs-930-02, University of Colorado (2002)
8. Dmitriev, M.: Profiling java applications using code hotswapping and dynamic call graph revelation. In: Proceedings of the Fourth International Workshop on Software and Performance, pp. 139–150. ACM, New York (2004)
9. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.P.: Traits: A mechanism for fine-grained reuse. *TOPLAS* 28(2), 331–388 (2006)
10. Ducasse, S., Pollet, D., Bergel, A., Cassou, D.: Reusing and composing tests with traits. In: Oriol, M., Meyer, B. (eds.) TOOLS EUROPE 2009. Lecture Notes in Business Information Processing, vol. 33, pp. 252–271. Springer, Heidelberg (2009)
11. Freedman, D., Pisani, R., Purves, R.: *Statistics*, 3rd edn. W. W. Norton & Company, New York (1997)
12. Gupta, A., Hwu, W.-M.W.: Xprof: profiling the execution of X Window programs. In: Proceedings of Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 1992/PERFORMANCE 1992, pp. 253–254. ACM, New York (1992)
13. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.* 20(4), 422–446 (2002)
14. Johnson, R.: TS: An optimizing compiler for Smalltalk. In: Proceedings of OOPSLA 1988, pp. 18–26. ACM, New York (1988)
15. Kleinrock, L., Muntz, R.R.: Processor sharing queueing models of mixed scheduling disciplines for time shared system. *J. ACM* 19(3), 464–482 (1972)
16. Mann, P.S.: *Introductory Statistics*. Wiley, Chichester (2006)
17. Martin, R.C.: *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, Englewood Cliffs (2002)
18. Meyer, M., Girba, T., Lungu, M.: Mondrian: An agile visualization framework. In: Proceedings of Symposium on Software Visualization (SoftVis 2006), pp. 135–144. ACM, New York (2006)
19. Miranda, E.: Brouhaha — A portable Smalltalk interpreter. In: Proceedings of OOPSLA 1987, pp. 354–365. ACM, New York (1987)
20. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Producing wrong data without doing anything obviously wrong! In: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, pp. 265–276. ACM, New York (2009)

21. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F.: Evaluating the accuracy of java profilers. In: Proceedings of PLDI 2010, pp. 187–197. ACM, New York (2010)
22. Mytkowicz, T., Diwan, A., Bradley, E.: Computers Are Dynamical Systems. *Chaos* 19, 033124 (2009)
23. Dain Samples, A., Ungar, D., Hilfinger, P.: SOAR: Smalltalk without bytecodes. In: Proceedings of OOPSLA 1986, pp. 107–118. ACM, New York (1986)
24. Tanter, É.: Execution levels for aspect-oriented programming. In: Proceedings of AOSD 2010, pp. 37–48. ACM, New York (2010)
25. Ungar, D., Patterson, D.: Smalltalk-80: Bits of History, Words of Advice Berkeley Smalltalk: Who Knows Where the Time Goes? In: Smalltalk-80: Bits of History, Words of Advice, pp. 189–206. Addison-Wesley, Reading (1983)
26. Yu, J., Yang, J., Chen, S., Luo, Y., Bhuyan, L.N.: Enhancing network processor simulation speed with statistical input sampling. In: Conte, T., Navarro, N., Hwu, W.-m.W., Valero, M., Ungerer, T. (eds.) HiPEAC 2005. LNCS, vol. 3793, pp. 68–83. Springer, Heidelberg (2005)

A Linear Regression Material

This section contains the relevant data and theoretical tools to construct the regression linear model described in Section 3.1 and Section 4.3.

Measurements. Table 2 lists 16 Pharo applications. Each of these applications covers a particular aspect of the Pharo library and runtime. *Collections* is an intensively used library to model collections. *Mondrian*, *Glamour* and *DSM* make an intensive use of graphical primitives and algorithms. *Nile* is a stream library based on Traits [9]. *Moose* is a software analysis platform which deals with large models and files. *Mondrian* and *Moose* heavily employ hash tables as internal representation of their models. *SmallDude*, *PetitParser*, *XMLParser* heavily manipulate character strings. *Magritte* and *Famix* are meta-models. *ProfStef* intensively makes use of reflection. *Network* uses primitive in the virtual machine. *ShoutTest* and *AST* heavily parse and manipulate abstract syntax trees. *Arki* is an extension of *Moose* that performs queries over large models.

These applications cover the features of Pharo that are intensively used by the Pharo communities: most of the applications are either part of the standard Pharo runtime or are among the 20 most downloaded applications. Not all the set of primitives offered by the virtual machines are covered by the applications. For example, none of them makes use of sound. We are not aware of any application that intensively uses Pharo’s musical support.

For each of these applications, we report the mean execution time over 10 trials to run its corresponding unit tests (**time taken (ms)**) and the number of sent messages (**# sent messages**). These reported results are averages over 10 runs. For each of these two measurements, we compute the standard deviations ($s_{messages}$ and $s_{TimeTaken}$; not reported here) and normalize it yielding $c_{time} = s_{TimeTaken} * 100 / TimeTaken$ and $c_{messages} = s_{messages} * 100 / messages$. These applications were run on a virtual machine modified to support our message counting mechanism.

Table 2. Applications considered in our experiment (second and third columns are average over 10 runs)

Application	time taken (ms)	# sent messages	$c_{time}\%$	$cmessages\%$
Collections	32 317	334 359 691	16.67	1.05
Mondrian	33 719	292 140 717	5.54	1.44
Nile	29 264	236 817 521	7.24	0.22
Moose	25 021	210 384 157	24.56	2.47
SmallDude	13 942	150 301 007	23.93	0.99
Glamour	10 216	94 604 363	3.77	0.14
Magritte	2 485	37 979 149	2.08	0.85
PetitParser	1 642	31 574 383	46.99	0.52
Famix	1 014	6 385 091	18.30	0.06
DSM	4 012	5 954 759	25.71	0.17
ProfStef	247	3 381 429	0.77	0.10
Network	128	2 340 805	6.06	0.44
AST	37	677 439	1.26	0.46
XMLParser	36	675 205	32.94	0.46
Arki	30	609 633	1.44	0.35
ShoutTests	19	282 313	5.98	0.11
<i>Average</i>			<i>13.95</i>	<i>0.61</i>

The source code of each of these applications is available online on Squeak-Source.

Estimating the sample regression line. For sake of completeness and providing easy-to-reproduce results, we provide the necessary statistical material. Complementary information may be easily obtained from standard statistical books [11].

For the least squares regression line $\hat{y} = a + b x$, we have the following formulas for estimating a sample regression line:

$$b = \frac{SS_{xy}}{SS_{xx}} \qquad a = \bar{y} - b \bar{x}$$

where \bar{y} and \bar{x} are the average of all y values and x values, respectively. The y variable corresponds to the **# sent messages** column and x to **time taken (ms)** in the table given above.

$$SS_{xy} = \sum xy - \frac{(\sum x)(\sum y)}{n} \qquad SS_{xx} = \sum x^2 - \frac{(\sum x)^2}{n}$$

where n is number of samples (i.e., 16, the number of applications we have profiled). SS stands for “sum of squares.” The standard deviation of error for the sample data is obtained from:

$$s_e = \sqrt{\frac{\sum SS_{yy} - b SS_{xy}}{n - 2}} \qquad \text{where } SS_{yy} = \sum y^2 - \frac{(\sum y)^2}{n}$$

In the above formula, $n - 2$ represent the degrees of freedom for the regression model. Finally, the standard deviation of b is obtained with $s_b = \frac{s_e}{\sqrt{SS_{xx}}}$.

Table 3. Cost of the Virtual Machine Modification

Application	Normal VM (ms)	Compteur (ms)	overhead (%)
Collections	32 317	32 323	0.02
Mondrian	33 719	33 720	0
Nile	29 264	29 267	0.01
Moose	25 021	25 023	0.01
SmallDude	13 942	13 944	0.01
Glamour	10 216	10 218	0.02
Magritte	2 485	2 485	0
PetitParser	1 642	1 642	0
Famix	1 014	1 015	0.1
DSM	4 012	4 013	0.02
ProfStef	247	247	0
Network	128	128	0
AST	37	38	2.7
XMLParser	36	36	0
Arki	30	30	0
ShoutTests	19	19	0

Table 4. Cost of the Application Instrumentation

Application	No inst (ms)	Inst. (ms)	overhead (%)
Collections	32317	33590	3.94
Mondrian	33719	36983	9.68
Nile	29264	36387	24.34
Moose	25021	26652	6.52
SmallDude	13942	24467	75.49
Glamour	10216	12976	27.02
Magritte	2485	4361	75.51
PetitParser	1642	2102	28.01
Famix	1014	3327	228.07
DSM	4012	4108	2.40
ProfStef	247	562	127.47
Network	128	875	583.87
AST	37	971	2524.00
XMLParser	36	559	1452.78
Arki	30	236	685.71
ShoutTests	19	40	111.76

Summarized Trace Indexing and Querying for Scalable Back-in-Time Debugging

Guillaume Pothier and Éric Tanter

PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile – Chile
www.pleiad.cl

Abstract. Back-in-time debuggers offer an interactive exploration interface to execution traces. However, maintaining a good level of interactivity with large execution traces is challenging. Current approaches either maintain execution traces in memory, which limits scalability, or perform exhaustive on-disk indexing, which is not efficient enough.

We present a novel scalable disk-based approach that supports efficient capture, indexing, and interactive navigation of arbitrarily large execution traces. In particular, our approach provides strong guarantees in terms of query processing time, ensuring an interactive debugging experience. The execution trace is divided into bounded-size *execution blocks* about which summary information is indexed. Blocks themselves are discarded, and retrieved as needed through *partial deterministic replay*. For querying, the index provides coarse answers at the level of execution blocks, which are then replayed to find the exact answer. Benchmarks on a prototype for Java show that the system is fast in practice, and outperforms existing back-in-time debuggers.

1 Introduction

Execution traces are a valuable aid in program understanding and debugging. Most research is centered on the capture of execution traces for offline automatic analysis [7,17,20]. However, there has been a recent surge of interest in *interactive* trace analysis through back-in-time, or omniscient, debuggers [5,8,9,10,11,12,13]. Such debuggers allow forward and backward stepping and can directly answer questions such as “why does variable x have value y at this point in time?”, thus greatly facilitating the analysis of causality relationships in programs.

The navigation operations provided by back-in-time debuggers are based on a small set of conceptually very simple queries. To achieve *interactive* navigation, those queries must execute extremely quickly, regardless of the size of the execution trace. It is therefore necessary to build and use indexes, otherwise queries would require scanning arbitrarily large portions of the execution trace. Interactive navigation in large execution traces requires an efficient indexing scheme tailored to the core set of back-in-time debugging queries:

Bidirectional stepping. These queries are similar to the usual stepping operations provided by traditional debuggers, with the added benefit of being able to perform them both *forward* and *backward* in time. *Step into* operations are very simple, as they consist in navigating to the next or previous event in the trace. *Step over* and *step out* operations, on the other hand, are more complex, as they require to skip all the events that occurred inside a method call. As the number of events to skip is potentially huge, it is not efficient to just perform a linear scan of the trace.

Memory inspection. Back-in-time debuggers support the inspection of the values of memory locations (such as object fields and local variables) at any point in time. To retrieve the value of a location at a particular point in time, the query to process consists in determining the last write operation to that location before the currently observed point. Again, as the last write can have happened much before the current observation point, it is not efficient to linearly scan the trace.

Causality links. Back-in-time debuggers support navigating via causality links, *e.g.* by instantly jumping to the point in time where a memory location was assigned its currently observed value. The corresponding query is actually the same as the one used to perform memory inspection: the last write operation to the location gives both the written value and the point in time at which it was written.

Interactive navigation in large execution traces is challenging: memory-based approaches allow fast navigation, but do not scale past a few hundred megabytes of trace data and therefore must discard older data [8,11]. To handle larger traces without losing information, a disk-based solution is mandatory [13], but this typically reduces the efficiency of the system. In addition, most back-in-time debuggers rely on directly capturing *exhaustive* executions traces [5,8,11,13]. Unfortunately, this incurs a significant runtime overhead on the debugged program, which is problematic for two reasons: (a) it makes the system less practical to use because of long execution times, and (b) the probe effect can perturb the execution enough that the behaviors to examine do not occur.

An alternative technique to avoid capturing exhaustive traces that alleviates the above issues is *deterministic replay* [13,15,16,19]. It consists in capturing only a *minimal trace* of non-deterministic events during the initial execution of a program. The minimal trace can then be deterministically replayed to obtain the exhaustive trace without affecting the execution of the debugged program. This is much cheaper than capturing an exhaustive trace, and thus greatly reduces the probe effect. Non-deterministic events are typically related to external inputs and system calls. However, another source of non-determinism is thread scheduling, something that is not properly supported in several deterministic replay systems.

Some deterministic replay systems support restarting the replay in the middle of the trace through *snapshots* that capture the state of the program at given points in time [15,16]. However these snapshots are *heavyweight* because they represent the full state of the heap. These snapshots can be produced efficiently by combining process forks and OS-level copy-on-write mechanisms, but they are

not easily serializable to disk. Therefore, snapshots remain in memory and older ones must be discarded, limiting the scalability or precision of the approach.

Contributions. This paper presents a novel scalable disk-based approach that supports efficient capture and interactive navigation of arbitrarily large execution traces. This approach relies on dividing the execution trace into bounded-size *execution blocks*, about which summary information is efficiently indexed. Execution blocks themselves are not stored on disk; rather, we support *partial deterministic replay*: the ability to quickly start replaying arbitrary execution blocks as needed. For querying, *summarized indexes* provide coarse answers at the level of execution blocks, which are then replayed and scanned to find the exact answer. More precisely:

- We describe the general approach and its instantiation as a new Java back-in-time debugging engine called STIQ, for Summarized Trace Indexing and Querying (Section 2). The approach is based on capturing non-deterministic events during the execution of the debugged program, followed by an initial replay phase during which snapshots are taken and indexes are constructed.
- We present an efficient deterministic replay system for Java (Section 3). This system supports partial deterministic replay through *lightweight snapshots* that are both fast to obtain and easy to serialize. We explain how these lightweight snapshots make it unnecessary to capture the heap.
- We propose indexing techniques for both control flow and memory accesses. The techniques leverage a recent succinct data structure [14] for efficient control flow indexing (Section 4), as well as the principle of temporal locality of memory accesses to reduce the amount of information to index (Section 5).
- We demonstrate through benchmarks that the approach enables a highly interactive back-in-time debugging experience (Section 6). Specifically, the proposed technique allows very fast index construction and query processing. Index construction takes 4 to 25 times the original, non-captured program execution time on realistic workloads. Query processing requires $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ CPU time for traces of arbitrary size n , and never exceeds a few hundred milliseconds in practice. We are not aware of any back-in-time debugging system that provides either such efficient index building, or such strong guarantees in query response time.

Finally, Section 7 discusses related work and Section 8 concludes.

2 Summarized Trace Indexing and Querying

Interactive back-in-time debugging requires that queries are processed fast enough to give the user a feeling of immediacy. For large execution trace, this mandates the use of indexing techniques: otherwise, arbitrarily large portions of the trace would have to be linearly scanned for each query. The system described in this paper, dubbed STIQ, provides an indexing scheme that is fast to build and yet processes queries very efficiently. The key insight is to divide the

execution trace into bounded-size execution blocks and to index only *summarized* information about each block; queries are then processed in two phases: the indexes first provide a coarse-grained answer at the level of execution blocks in $\mathcal{O}(\log n)$ time, and the relevant execution block is then scanned to find the exact answer in $\mathcal{O}(1)$ time (as the size of blocks is bounded).

This section gives an overview of the complete process, whose steps are detailed in subsequent sections, and presents the overall system architecture.

2.1 Process Overview

The STIQ process consists of four phases: trace capture, initial replay, summarized indexing, and querying.

Trace capture. The debugged program is transparently instrumented so that whenever a non-deterministic operation (such as a system call or a memory read) is executed, its outcome is recorded into a *minimal execution trace* that is stored on disk. The trace is interspersed with regular synchronization points that give a rough timestamping of events, so that an approximate ordering of events of different threads can be obtained so as to resolve race conditions.

Initial replay. Although the minimal trace produced by the capture phase is sufficient to deterministically replay the debugged program, it is not directly useful for our indexing process: it contains memory read events, whereas the memory writes are those that must be indexed. An *initial replay* is thus performed to obtain a *semi-exhaustive trace* consisting of memory write events and cursory method call information (only the fact that a method is entered/exited is needed). This is achieved by feeding the minimal trace to a *replayer* that re-executes the original program, but with non-deterministic operations replaced by stubs that read the recorded outcome from the trace. The program is also instrumented so that it generates the needed semi-exhaustive trace. Additionally, when a synchronization point is encountered, a *lightweight snapshot* is generated so that replay can be restarted from that point later on. Snapshots thus define the boundaries of individually replayable *execution blocks*.

Summarized indexing. The semi-exhaustive trace produced in the initial replay is not stored but rather consumed on the fly by an *indexer* that efficiently builds the indexes. The *indexer* summarizes the information of each execution block, as depicted in Figure 1. For method entry and exit events, the indexer builds a control flow tree and represents it as a Range Min Max Tree (RMM Tree) [14], a state-of-the-art succinct data structure that allows very fast navigation operations. Auxiliary structures map the beginning of execution blocks to positions in the RMM Tree. Together, these structures allow efficient stepping operations while using only slightly more than one bit per event. For memory writes, the indexer coalesces all the writes to a given location that occur within an execution block into a single index entry. In practice, this reduces the number of entries to index by 95%: because of

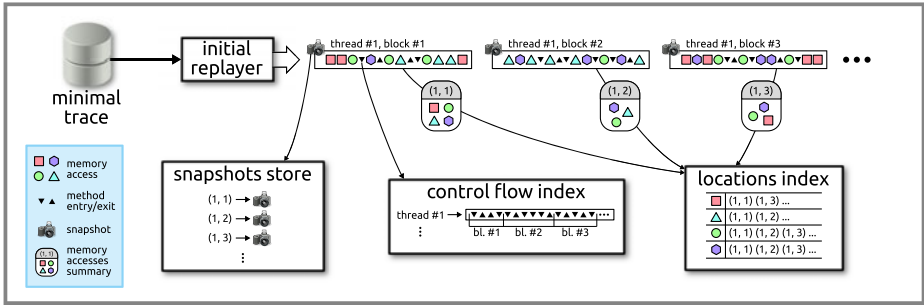


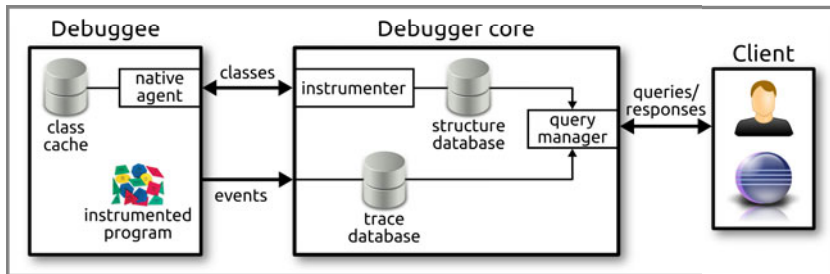
Fig. 1. Summarized indexing process

the principle of temporal locality, if a memory location is accessed at some point in time, it is very likely that it will be accessed again in the near future, *i.e.* in the same execution block. Finally, snapshots are simply stored in an on-disk dictionary structure.

Querying. The indexes constructed in the previous step can determine the execution block that contains the answer to a given query very quickly: they only require $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ CPU time (with very favorable hidden constants—in practice they take 1-10ms). Once the execution block has been determined, the corresponding snapshot is retrieved (again in $\mathcal{O}(\log n)$ disk accesses) and the block is replayed and then scanned to find the exact event of interest in $\mathcal{O}(1)$ CPU time (as the size of execution blocks is bounded and thus does not depend on the size of the trace). In practice, queries take a dozen milliseconds on average, and never take more than a few hundred milliseconds (see Sect. 6).

2.2 System Architecture

Our system uses an out-of-process database to store and index the execution trace. The overall architecture is depicted below:



It consists of three elements:

1. The *debuggee*, which is the Java VM that executes the program to debug. It contains a special native agent that intercepts class loading so that classes

are instrumented prior to execution (either by sending their bytecode to an out-of-process instrumenter, or by looking them up in a class cache if they have already been instrumented in a previous session). When executed, instrumented code emits events that are sent to the debugger core.

2. The *debugger core*, which consists in (a) an instrumenter that receives the original classes from the debuggee and inserts the event emission code before sending the modified classes back to the debuggee, (b) a structure database that records information about the classes, methods and fields of the debugged program, (c) a trace database that stores and indexes the events emitted by the debugged program, and (d) a query manager that uses the database indexes to quickly answer queries.
3. The *client*, which is the user interface of the debugger. It presents the user with views over the debugging session and controls to interactively navigate in the execution trace using back-in-time debugging metaphors: stepping backward and forward, navigating runtime data dependencies, etc.

3 Trace Capture and Partial Deterministic Replay

This section describes the key features of our deterministic replay system. Many implementation details are omitted or only glossed over. Section 3.1 describes how the trace is captured: which events are captured, how we avoid having to simulate the heap, and how memory locations are identified. It also describes the scoping abilities of our system. Section 3.2 discusses the replayer, and Section 3.3 details how and when snapshots are taken.

3.1 Capture

Trace capture is achieved by transforming the original program through bytecode instrumentation so that non-deterministic events are serialized and stored when the program is executed.

Non-deterministic events. Non-deterministic operations are those whose outcome can vary from one program execution to another, and thus must be recorded so as to enable deterministic replay. These operations are:

- *Native operations.* The outcome of native operations such as disk or network reads cannot be predicted. In addition, as discussed later in this section, our system supports user-defined scoping. Out-of-scope methods are considered non-deterministic.
- *Heap memory reads.* Thread scheduling can affect the order in which memory write operations are executed, and as scheduling is outside the control of the debugged program, the contents of memory is non deterministic.¹

¹ In the case of Java, only the contents of heap memory is non-deterministic, as the virtual machine does not allow concurrent access to stack memory.

Dealing with memory non-determinism. A strategy to deal with the non-determinism of memory reads due to multi-threading consists in recording the order in which threads are scheduled, and forcing the same order during replay [3]. This is of limited usefulness with multicore architectures, however, as in this case concurrency occurs at the hardware level. Another strategy, which we use in our system, consists in recording the value obtained by every memory read [1].

Avoiding heap simulation. Although capturing memory reads is enough to allow a fully accurate replay, it still requires to simulate the state of the whole heap during replay because some control-flow-altering operations (polymorphic method dispatch and `instanceof`) rely on the content of the heap, as the type of objects must be known. The simulated heap would occupy as much memory as the heap of the original program.

Fortunately, it is possible to completely avoid simulating the heap by recording the outcome of the above control-flow-altering operations, even though they are deterministic. This has a very small impact on the trace capture overhead, but drastically reduces the memory requirements of the system, thus freeing valuable memory for the indexing process. Moreover, as the heap is not needed anymore for replay, the only information needed to start replaying at arbitrary execution block boundaries can be represented in *lightweight snapshots* that only contain the values of the local variables of the current stack frame and the identifier of the current method. Such snapshots are cheap to obtain and take up very little space.

Identification of memory locations. The reconstitution of program state at arbitrary points in time requires the indexing of memory locations; it is therefore necessary to be able to uniquely identify each memory location. Two distinct types of locations must be considered: heap locations (object fields and array slots), and stack locations (local variables).

For heap locations, we regard both objects and arrays as *structures* that contain a fixed number of slots. Structures are assigned a unique id at creation time, and the id of a particular location within a structure is obtained by adding the index of the slot to the id of the structure. For objects, the index of the accessed slot is determined statically (each field of a given class can be assigned an index statically). For arrays, the index of the accessed slot is explicitly specified at runtime. To ensure the uniqueness of memory location ids, the sequence that is used to give a new structure its unique id is incremented by the number of slots of the structure.

In Java, the ideal way to store the id of structures would be to add a field to the `Object` class. However, adding fields to certain core classes such as `Object`, `String` or array classes is problematic in most Java implementations (*e.g.* doing so makes the HotSpot JVM crash). To solve this issue, we add the id field to all non-problematic subclasses of `Object`, and we use a global weak identity hash map for the problematic classes; this unfortunately incurs a significant runtime overhead (as shown in Sect. 6).

For stack locations, we use a compound id consisting of the id of the current thread, the current call stack depth, and the index of the variable within the stack frame. This scheme does not uniquely identify each location, because local variables in subsequent invocations of different methods by the same thread at the same depth will share the same id. However, this is not a problem because the temporal boundaries of method invocations are known. We come back to this issue in Section 5.3.

Scoping. In many cases some parts of the debugged program might be trusted to be free of bugs (for instance, the JDK classes in the case of Java), or the bug can be known to manifest only under certain runtime conditions [13]. Trace scoping reduces the runtime overhead on the debugged program, the size of the execution trace, and the indexing and querying cost, by limiting the set of events that are captured. Static scoping consists in limiting capture to a set of classes, while dynamic scoping consists in activating or deactivating capture dynamically at runtime. Our system currently supports only static scoping; dynamic scoping would however be relatively easy to add.

The user configures the static scope by specifying a set of classes or packages to include or exclude from the trace. We define the set of out-of-scope methods as all the regular bytecode-based methods that belong to out-of-scope classes, as well as all native methods.

By definition, the execution of out-of-scope code cannot be replayed. It is therefore necessary to capture additional information at the runtime boundaries between in-scope and out-of-scope code. In particular, the return values of out-of-scope methods called by in-scope methods, as well as the arguments of in-scope methods called by out-of-scope code must be captured.

Unfortunately, because of polymorphism it is not possible to statically determine whether a particular call site will result in the execution of an in-scope or of an out-of-scope method; similarly, it is not possible to determine if a given method will be called by in-scope or out-of-scope code. Therefore, in the trace capture phase we instrument the *envelope* (ie. entry and exit) of all out-of-scope methods in order to maintain a thread-local *scope stack* of booleans that indicates whether the thread is currently executing in-scope or out-of-scope code. Whenever the execution of an in-scope method starts, the top of the stack is checked to decide if method arguments must be captured; similarly, whenever an out-of-scope method exits, the top of the stack is checked to decide if the return value should be captured.

3.2 Initial Replay

The main task of the replayer is to inject the recorded outcomes of non-deterministic operations into the replayed program. To that end, we transform the program through bytecode instrumentation so that non-deterministic operations are replaced by proxies that read their outcome from the trace.

As explained above, the heap is never explicitly reconstituted; therefore, the replayer never needs to instantiate any class of the original program: instances

are instead represented by a generic `ObjectId` class that is simply a container for the identifier of the object². All the non-static in-scope methods of the program are replaced by static ones that take an additional `ObjectId` parameter that represents the target of the method.

On the other hand, as out-of-scope methods do not record any information in the trace (except the envelope as explained above), they all behave exactly in the same way as far the replayer is concerned: a black box that consumes parameters and generates a return value. Therefore the original out-of-scope methods are not used at all in the replayer, and are collectively replaced by a single, generic method provided by the replayer infrastructure.

3.3 Snapshots

Snapshots define the boundaries of execution blocks. Recall that snapshots are taken during the initial complete replay of the program, and not during capture, so as to reduce the runtime overhead of capture as much as possible. We now describe how and when snapshots are taken.

Snapshot probes. The ability to take a snapshot at a given program point requires the insertion of a piece of code, called a *snapshot probe*, that performs the following tasks:

1. Check if a snapshot is actually requested at this moment, by reading a thread-local flag (detailed below).
2. Store the necessary information in the snapshot: identification of the snapshot probe, current position in the minimal execution trace, and the values of local variables and operand stack slots.

Recalling that the heap is not reconstituted during replay, the information mentioned above is sufficient for replaying the current method and all the methods called from there, recursively. It is not sufficient, however, to return to the caller of the current method: the stack frame of the caller is not recorded in the snapshot. This problem is addressed by always inserting snapshot probes after method calls, and forcing the creation of a snapshot at those probes if a snapshot was taken during the execution of the method. Thus, although the partial replay cannot directly continue after the current method returns, there is always another snapshot at the right point in the caller method so that another partial replay can be started right where the previous one finished.

Snapshot intervals. The size of execution blocks must be chosen considering a tradeoff between indexing efficiency and querying efficiency:

- Larger blocks make it possible to coalesce more object accesses into one index entry, thus increasing indexing throughput.
- Shorter blocks can be replayed faster and thus queries can be answered faster.

² We use a container instead of a scalar because the actual value of the id is mutable in the case of instantiations, but this is beyond the scope of this paper.

It is important to take into account the involved magnitudes:

- Indexing is performed on the fly during the initial complete replay, and preemptively considers all of the objects that exist during the execution of the program: all object accesses in the trace incur an indexing cost. Therefore, small variations in indexing throughput can noticeably affect the global efficiency of the system.
- Queries deal with individual objects and are performed by a human being, who cannot differentiate between a one microsecond or a one millisecond response time. Therefore, important variations in querying efficiency can go largely unnoticed up to a certain point.

The time interval between snapshots define the maximum size of execution blocks³. This interval is configurable by the user, controlling the tradeoff between indexing efficiency and query response time.

Probe density. Probes should be inserted densely enough in the program so that a snapshot can be taken quickly once it is requested. However, snapshot probes are costly both in code size and in speed (because of the runtime check) so it is preferable to limit their number. As we must insert snapshot probes after every method call anyway (as explained above), the density is usually already sufficient with just those probes. Nevertheless, it is possible for the program to contain a loop with no method calls at all, like a complex calculation on a large array; in this case, an additional probe would be needed inside the loop. For the sake of simplicity, and because this kind of program is rather infrequent, we currently do not insert these additional probes.

4 Indexing of Control Flow

We now turn to the indexing techniques. This section describes the indexing of control flow, and Section 5 describes the indexing of memory accesses. While *step into* queries simply consist in moving to the next/previous event in the execution trace, efficiently executing *step over* and *step out* queries requires an index: otherwise it would be necessary to linearly scan the execution trace to skip the events that occurred within the control flow of the stepped over call, or between the current event and the beginning of the current method.

The control flow can be represented as a tree whose nodes correspond to method calls. Stepping operations then simply correspond to moving from a node to its next/previous sibling (for *step over*), or to its parent (for *step out*). We store the control flow tree using a *Range Min-Max Tree* (RMM Tree) [14], a recent *succinct data structure* that is disk-friendly, fast to build and supports fast navigation operations. Auxiliary data structures maintain a correspondence between execution blocks and their initial node in the control flow tree so that

³ We also set a minimum size for execution blocks, so that a thread that spends most of its time sleeping does not generate plenty of useless snapshots.

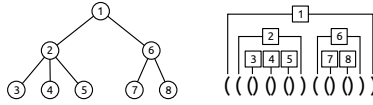


Fig. 2. A tree and its balanced parentheses representation

the block corresponding to a given node can be determined during queries. This approach uses only slightly more than 1 bit per method call or return event, while requiring only a few milliseconds to answer arbitrary stepping queries, making them seem instantaneous to the user.

This section first briefly describes the RMM Tree structure and then explains our control flow indexing and querying mechanism.

4.1 Range Min-Max Tree

A succinct data structure is one that stores objects using space close to the information-theoretic lower bound, and at the same time supports fast queries on the stored objects. In the case of a tree⁴ with n nodes, the lower bound is $2n - \Theta(\log n)$ bits [14]. A classical way to represent trees using $2n$ bits is the *balanced parentheses sequence* (see Figure 2): each node is represented by a pair of matched parentheses that enclose the representation of its children. A node in the tree is identified by the position of the corresponding opening (or closing) parenthesis.

Although such a structure is compact (as only one bit is needed for each parenthesis), it does not allow *per se* to quickly answer queries like finding the next sibling, previous sibling or parent of a given node. The RMM Tree [14] adds an indexing layer on top of the balanced parentheses representation that incurs very little space overhead while allowing extremely fast querying. In theory, the RMM Tree supports queries in constant time $\mathcal{O}(c^2)$ with a data structure using $2n + \mathcal{O}(n/\log^c n)$ bits, for any constant $c > 0$. In practice, we trade the constant time for logarithmic time with a very big base.

The essential idea of the RMM Tree is to compute a running sum of the bits that represent the parentheses sequence: opening parentheses increment the sum by 1, and closing parentheses decrement the sum by 1. For each fixed-size block of parentheses, a summary indicating the minimum and maximum value that the sum takes within the block is stored separately. Fixed-size blocks of summaries are then recursively summarized (the minimum and maximum of a whole block of summaries are separately stored at a higher level). This results in a tree structure of height H in which the leaves are the bits that represent the balanced parentheses sequence, and the nodes contain the minimum and maximum value of the running sum in their subtree. Subtle observations about the relationship between the running sum and the primitive tree navigation operations make it possible to guarantee that queries can be answered by accessing at most $2H$ blocks (going up to the root and then down to the correct leaf) [14].

⁴ Specifically, ordinal trees, where a node can have any number of ordered children.

Algorithm 1. Find return event.

Finds the return event corresponding to the call event denoted by (t, b, i) .

```

1: function FINDRETURN( $t, b, i$ )
2:    $tree \leftarrow getCFlowTree(t)$ 
3:    $p_{call} \leftarrow eventToPosition(t, b, i)$ 
4:    $p_{ret} \leftarrow tree.getClose(p_{call})$ 
5:    $(t_{ret}, b_{ret}, i_{ret}) \leftarrow positionToEvent(t, p_{ret})$ 
6:   return  $(t, b_{ret}, i_{ret})$  ▷ By construction  $t = t_{ret}$ 
7: end function

```

Algorithm 2. Event to position.

Returns the RMM Tree position corresponding to the given event reference.

```

1: function EVENTTOPOSITION( $t, b, i$ )
2:    $(tree, map) \leftarrow getCFlowIndex(t)$ 
3:    $p \leftarrow map.getPos(b)$ 
4:    $block \leftarrow getBlock(t, b)$ 
5:   for  $k$  in  $1, i$  do
6:     if  $block[k]$  is a call or return event then
7:        $p \leftarrow p + 1$ 
8:     end if
9:   end for
10:  return  $p$ 
11: end function

```

In practice, blocks correspond to disk pages (usually 4KB). The summary information to store for each block (minimum and maximum values plus some ancillary data) occupies only 10 bytes. As a consequence the tree is quite flat: for instance, an RMM Tree of height 4 can store up to $\lfloor \frac{4096}{10} \rfloor^3 \cdot 4096 \cdot 8 \simeq 2 \cdot 10^{12}$ bits in its leaves and occupies around $4096 \cdot \sum_{i=0}^3 \lfloor \frac{4096}{10} \rfloor^i \simeq 280\text{GB}$, thus requiring roughly 2.005 bits per original tree node (slightly more than 1 bit per parenthesis).

4.2 Indexing and Querying

The indexing process for control flow is straightforward: each execution thread has its own RMM Tree that stores all the method call (resp. return) events as one opening (resp. closing) parenthesis as they occur. Also, execution blocks are identified by a thread-local block id, equal to the timestamp of the initial snapshot of the block. Blocks ids are unique within a thread, but not globally. Whenever a new execution block starts, a pair (block id, current RMM position) is stored in a bidirectional map, which makes it possible to either retrieve the block id given a RMM position, or the RMM position given a block id. More precisely, this bidirectional map consists of two BTrees, one where the block ids are the keys and the RMM positions are the values, and another one with the opposite relationship. As BTrees use binary search for keys, the keys used for lookup do not need to be exact values. We take advantage of this feature when looking up a block id given a position: there is usually no record for the exact position, but we can instead return the id of the block that contains this position.

Algorithm 3. Position to event.

Returns the event reference corresponding to the given RMM Tree position.

```

1: function POSITIONTOEVENT( $t, p$ )
2:    $(tree, map) \leftarrow getCFlowIndex(t)$ 
3:    $b \leftarrow map.getBlockId(p)$ 
4:    $p_0 \leftarrow map.getPos(b)$   $\triangleright p_0$  is the position of the RMMTree corresponding to the beginning
   of block  $b$ 
5:    $block \leftarrow getBlock(t, b)$ 
6:    $i \leftarrow 1$ 
7:   while  $p_0 < p$  do
8:     if  $block[i]$  is call or return event then
9:        $p_0 \leftarrow p_0 + 1$ 
10:    end if
11:     $i \leftarrow i + 1$ 
12:  end while
13:  return  $(t, b, i)$ 
14: end function

```

To perform a step over operation⁵, it is necessary to determine the return event corresponding to the call event that is being stepped over. The result of the step over operation is simply the event following the return event. The *findReturn* function (Algorithm 1) is thus the basis of the step over operation.

Events are identified by a (t, b, i) tuple where t is the thread id, b is the block id, and i is the index of the event within the block. The algorithm consists of three steps: (a) determining the position of the bit (or opening parenthesis) of the RMM Tree that corresponds to the given method call event, (b) determining the corresponding closing parenthesis, that corresponds to the return event, and finally (c) translating the RMM Tree position back to an event reference. Translating back and forth between event references and RMM Tree positions is implemented in the subroutines specified in Algorithms 2 and 3.

The algorithms use the following auxiliary procedures:

- *getBlock*(t, b) replays block b of thread t and returns the exhaustive list of events for that block.
- *getCFlowIndex*(t) returns the RMM Tree and bidirectional map corresponding to thread t ; *getCFlowTree*(t) returns only the RMM Tree. These are constant-time operations.

There are three components to the cost of the algorithm:

- The replaying of the initial and final execution blocks (although the initial execution block is usually available in a cache, as it corresponds to the events the user was currently observing). These operations take a time proportional to the size of the blocks, which is a constant that can be tuned by the user.
- The obtention of block ids and positions through the bidirectional map.⁶ These operations are BTree lookups that require $\mathcal{O}(\log n)$ disk accesses.
- The navigation to the closing parenthesis in the RMM Tree. This operation also requires $\mathcal{O}(\log n)$ disk accesses.

⁵ We describe forward step over; backward step over and step out are similar.

⁶ In Algorithm 3, lines 3 and 4 are actually a single operation, as the binary search for the given position gives both the registered position and the corresponding block id.

In practice, arbitrary stepping queries only take a dozen milliseconds on average, and never take more than a few hundred milliseconds, allowing highly interactive stepping (see Sect. 6).

5 Indexing of Memory Accesses

Two of the essential features of back-in-time debuggers are the ability to inspect the state of memory locations at any point in time, and the ability to instantly navigate to the event that assigned its value to a location. Both features rely on the same basic query: finding the last write to the location that occurred before a reference event (the point of observation). The write event indicates both the value that was written and the moment it was written⁷.

The key to being able to answer such queries efficiently is to have a separate index for each memory location; if a single index is shared between several locations, a linear scan of the index (which can take a time proportional to the size of the trace) is necessary. This said, constructing an exhaustive index of all write accesses for each location is prohibitively costly [13]. Instead, we index only a *summary* of the write accesses: we coalesce all accesses to a given location that occur within an execution block to a single index entry. We thereby exploit the principle of *temporal locality*: if a given location is accessed at a point in time it is very likely to be accessed again in the near future, *i.e.* in the same execution block. In practice, this approach allows us to discard around 95% of memory accesses. This compression ratio, along with the pipelined index construction process described later, makes it possible to maintain a separate index for each memory location.

To answer queries, the index is used to determine, in logarithmic time, the execution block that contains the access of interest; the block is then replayed and linearly scanned to retrieve the exact event. As block size is bounded, this linear scan does not depend on the size of the trace, and is very fast in practice, as will be shown in Sect. 6.

In the following we first present the general structure of the index and the way it is queried, before explaining how to build it efficiently using a multicore-friendly pipelined process. This section deals mostly with heap memory locations (object fields and array slots). The capture system assigns a unique identifier to each heap location, as explained in Sect. 3.1. We explain how stack locations (local variables) are handled in Sect. 5.3.

5.1 Index Structure and Querying

Memory inspection queries consist in finding the last write to a given location that occurred before a certain reference event. As explained above, there is one

⁷ Although to simplify the presentation we consider a single result for memory inspection queries, there is actually a *set* of write events that *might have written* the current value of the location at the time the reference event occurred. The reason the query produces a set and not a single event is that the resolution of data races is limited by the accuracy of the timestamping of events.

Algorithm 4. Memory location reconstitution.

```

1: function GETLASTWRITE(loc, (t, b, i))
2:   index  $\leftarrow$  getLocationIndex(loc)
3:   (b2, threads)  $\leftarrow$  index.getAtOrBefore(b)
4:   for t2 in threads do
5:     block  $\leftarrow$  getBlock(t2, b2)
6:     if b2 = b and t2 = t then
7:       limit  $\leftarrow$  i - 1
8:     else
9:       limit  $\leftarrow$  length(block)
10:    end if
11:    for k in limit, 1 do
12:      if block[k] is write to loc then
13:        yield (t2, b2, k)
14:        break
15:      end if
16:    end for
17:  end for
18: end function

```

individual index for each memory location. As there are many such location indexes, there is also a master index used to retrieve particular location indexes.

The process of answering a query is sketched in Algorithm 4. It consists of three main steps:

1. Retrieve the index for the particular location using the master index (line 2). This is implemented as a BTree lookup, and thus requires $\mathcal{O}(\log n)$ disk accesses.
2. Within the location index, search the execution block(s) that occurred at the same time or just before the block *b*, which contains the reference event (line 3). This search can produce as many blocks as there are threads writing to the location in the same time span as block *b*. As we explain later, there are different implementation of the location indexes, according to the number of entries in the index, but in the worst case the search requires $\mathcal{O}(\log n)$ disk accesses.
3. Replay the blocks of the previous step to find the last write(s) to the inspected location. Although there can be any number of blocks to replay, the size of blocks decreases with the number of concurrent threads. This is because blocks are delimited by elapsed time (see Sect. 3.3): the more threads execute concurrently at a given time, the less events there are in the corresponding blocks. The time required to replay those blocks is therefore bounded and does not depend on the size of the trace.

As shown in Sect. 6, such queries in practice only take two dozen milliseconds on average, and never more than a few hundred milliseconds, allowing very fast reconstitution of memory locations.

5.2 Pipelined Index Construction

The previous section showed that it is possible to query the memory locations index in logarithmic time. We now show that the index can also be efficiently

⁸ Modulo the number of available CPU cores, but this is also a constant.

built. As explained in Sect. 2, an initial replay of the minimal trace is performed so as to obtain a semi-exhaustive execution trace that contains memory write events. The semi-exhaustive trace is consumed on the fly by the indexer.

The indexing process is divided into 5 pipelined stages (see Fig. 3), and can thus take advantage of multicore systems, as the different stages can run in parallel (although the CPU utilization is not evenly distributed between all stages). The first three stages operate in main memory, while the latter two deal with storing data on disk. By conveniently ordering the data, the first stages help reduce the amount of disk seeks needed at the later stages.

Summarizing. This stage (Fig. 3a) is instantiated for each thread of the debugged program. It scans incoming execution blocks, and for each memory write, it adds the identifier of the written location to a hash set. Using a set is key to our indexing approach, ensuring that each written location appears only once per execution block. Once an execution block is finished, the set is transformed into a (t, b, a) tuple where t and b are the thread and block id, and a is a sorted array of the location identifiers that have been written to within the block. The tuple is then passed on to the next stage.

Because execution blocks are relatively small in practice, all the operation of this stage can be performed in memory.

Reordering. During trace capture, events are stored in thread-local buffers before being stored in the minimal trace. Busy threads emit many events, so they quickly fill their event buffers, while threads that spend a lot of time waiting might take a long time to fill a single buffer. It is therefore possible that execution blocks of different threads are stored in the trace out of order. The later stages of the pipeline can cope with this situation, but at the cost of a significant loss of throughput. The goal of the reordering stage (Fig. 3b) is thus to avoid as much as possible the costly reordering by downstream stages.

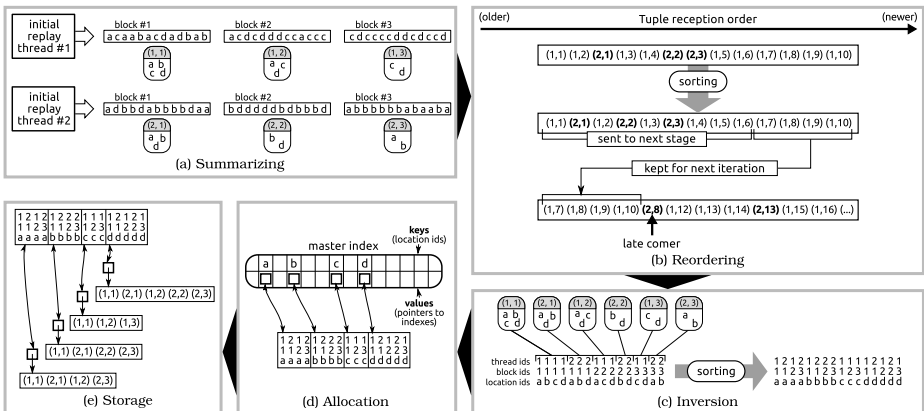


Fig. 3. The five stages of the indexing pipeline

This stage accumulates the tuples in a buffer, and when their total size exceeds a certain threshold (32 MB in practice), they are sorted by block id, and the oldest ones (the oldest 60% in practice) are passed on to the next stage in a bundle for processing. The remaining ones stay in the buffer and will be sorted again, along with newer ones and possible late comers, in the next round. The aforementioned threshold size is chosen to be small enough so that the data sets of this stage and the following one can fit in main memory, but large enough to impede most out-of-order blocks from going through.

Inversion. This stage (Fig. 3e) receives bundles of (t, b, a) tuples and operates in two phases:

1. Each (t, b, a) tuple is expanded into a list of (t, b, l) tuples, one for each memory location $l \in a$. The threshold size chosen in the previous stage has to be small enough that the expanded tuples of this stage can fit in main memory.
2. The concatenated list of all the (t, b, l) tuples is then sorted by location id l , then by block id b and finally by thread id t .

As a consequence of the sorting, the tuples produced in this stage are grouped by location, which reduces the amount of disk seeks needed to build the on-disk index in the following stages. Additionally, having the tuples within each group sorted by block id and thread id enables the use of compact encodings, thus reducing the size of the indexes, as explained below.

Allocation. For each location group in the tuple list produced by the previous stage, an entry is allocated in the master index (or retrieved, if it already existed). An entry is simply a pointer that references the page where the individual index corresponding to the location is stored. The tuple list of the previous stage is passed on to the next stage, along with a list of allocated entries, so that the next stage can perform the actual storage of the tuples of each group without having to access the master index anymore.

Storage. This final stage performs the actual storage of (t, b) tuples in the individual indexes corresponding to each location l . According to the number of tuples to store in each index, three different index formats are used:

- Because most objects are short lived and therefore are accessed in only one execution block, most indexes (around 80%) contain a single tuple. We store these indexes in shared pages, which we call *singles pages*. Thanks to the ordering performed in the previous stage and the use of gamma codes⁹ to store the difference between successive tuple components, a 4KB singles page contains around 800 indexes on average.

⁹ Gamma codes [4] represent an integers x in (roughly) $2 \log_2 x$ bits. Small numbers are thus encoded in very few bits.

- For indexes that contain more than one tuple but less than the number of tuples that can fit in half a disk page, we use another type of shared pages, which we call *n-shared pages*, with $n \in \{2^m\}$ for $m \in [1..7]$. In these pages, space is evenly distributed between n indexes.
- For bigger indexes, we use BTrees where keys are block ids and values are thread ids. Again, we use gamma codes to store the tuples in these trees.

As indexes are built on the fly, we do not know beforehand what the size of each index will be. Indexes thus migrate from singles page to n -shared pages to BTrees as more tuples are added.

5.3 Local Variables

Having a separate index for each memory location implies that each location can be uniquely identified. As explained in Section 3.1, our trace capture system assigns a unique id to each heap location (object fields and array items), but this uniqueness constraint is relaxed for stack locations (local variables). Stack locations are assigned a compound id that is made of the thread id, the local variable index, and the call stack depth. This entails that there cannot be a separate index for each stack location, as the stack frames of subsequent method executions at the same level will share some local variable indexes. However, queries can still be processed efficiently: we already know the temporal boundaries during which particular stack locations exist (these boundaries are defined by method entry and exit, which are indexed). To process a stack location inspection query, we query the corresponding index as if it was not shared. If the answer is outside the temporal boundaries of the current method invocation, it means there is no write to the variable before the reference event.

6 Benchmarks

In this section we present the experimental results we obtained with our STIQ system, and we compare them with those obtained with TOD [12,13], our previous disk-based back-in-time debugger for Java, which to the best of our knowledge still represents the state of the art up to now. (We compare to other related systems in Section 7.) All the benchmarks were performed on a Quad-core 2.40GHz Xeon X3220 machine with 4GB RAM and two 160GB SCSI hard drives in a RAID-0 configuration, running the x86_64 Linux 2.6.24 kernel. We used the Sun HotSpot 1.6.0_22 32 bits JVM in server mode for both the debuggee program and the indexing server.

We used the *avro* and *lusearch* benchmarks of the DaCapo v9.12 benchmark suite [2], as well as a toy benchmark called *burntest* that stresses STIQ capture and indexing by performing almost only method calls and field accesses (it consists in repeatedly navigating a large in-memory tree). For DaCapo benchmarks, we use the `small` dataset size, and force two driver threads. For both STIQ and TOD, the JDK classes were configured to be out of scope.

We first present global results (capture overhead, indexing speed and query efficiency) that show the competitiveness of our approach. We then give a detailed accounting of the time and space resources needed for individual features.

6.1 Global Results

Table 1 shows the impact of trace capture on the debugged program. It varies between 10x and 30x for STIQ and between 22x and 176x for TOD¹⁰. The overhead of STIQ is much lower than that of TOD, as well as that of other back-in-time debuggers: the Omniscient Debugger [8] has an overhead of around 120x, while Chronicle (discussed in Sect. 7) reports a 300x overhead. Also, STIQ has an overhead comparable with other deterministic replay systems like Nirvana [1], which reports a 5x-17x overhead. Nirvana however is only concerned about deterministic replay, not trace indexing.

Table 1. Runtime overhead of trace capture

Workload	t_0	STIQ		TOD	
		t_{STIQ}	o_{STIQ}	t_{TOD}	o_{TOD}
avrora	5.5s	163s	30x	968s	176x
lusearch	7s	69s	10x	157s	22x
burntest	5.2s	65s	12x	427s	82x

t_0 : original execution time without trace capture

t_{TOD} , t_{STIQ} : execution time with trace capture

o_x : runtime overhead (t_x/t_0)

With respect to trace capture, even though the numbers are comparatively favorable to STIQ, the capture overhead still remains high; further effort is necessary in this regard.

Table 2. Replay and indexing time (and ratio to original execution time)

Workload	STIQ			TOD
	replay	indexing	total	
avrora	95s (17x)	46s (8.4x)	141s (25x)	152min (1664x)
lusearch	19s (2.7x)	13s (1.8x)	32s (4.5x)	16min (138x)
burntest	39s (7.5x)	375s (72x)	414s (80x)	52min (606x)

Table 2 indicates the time needed to index the captured traces. For the Da-Capo benchmarks, STIQ actually uses less time to perform the initial replay and build the indexes than to capture the trace. For burntest on the other hand, the

¹⁰ This shows that the published worst-case runtime overhead of 80x for TOD [13] was not actually the worst case.

Table 3. Space usage

Workload	STIQ		TOD	
	trace	index	trace	index
avroa	5GB	0.27GB	35GB	65GB
lusearch	1.1GB	0.16GB	6.2GB	11.3GB
burntest	1.5GB	2.7GB	20GB	39.4GB

Table 4. Average (and maximum) query response time

Workload	STIQ		TOD	
	stepping	memory	stepping	memory
avroa	<1ms (0.24s)	19ms (0.5s)	12ms (6.8s)	41s (96min)
lusearch	<1ms (0.37s)	27ms (0.48s)	5.2ms (1.6s)	1.9s (4min)
burntest	6.9ms (0.65s)	8.6ms (0.17s)	17ms (0.74s)	3.4s (17min)

indexing is very slow, as that workload consists only in method calls and field accesses, with no extra deterministic computation in between. STIQ is (at least) one order of magnitude faster than TOD to build the indexes.

Table 3 shows the size of the captured execution traces, as well as the size of the created indexes. STIQ traces are much smaller than those of TOD, showing the benefit of using a deterministic replay system versus exhaustive trace capture. It is notable that for the DaCapo benchmarks, STIQ produces indexes that are much smaller than the trace itself; for `burntest` the index is almost twice as big as the trace, again because `burntest` is all about method calls and field accesses, which are the two kinds of events that are indexed. Also worthwhile to note is the fact that TOD indexes are always bigger than the already bulky traces.

Table 4 shows the query response time of STIQ and TOD. For stepping queries, we divide each thread of the execution trace into 100 equal intervals and starting at the beginning of each interval we alternately perform step over and step out operations until the root of the control flow is reached. As we get closer to the control flow root, step over operations must skip a greater number of events. For memory inspection queries, we first realize a (non-timed) pass that collects the locations to inspect: we divide each thread into 20 equal intervals and start scanning the trace at the beginning of each interval, collecting accessed locations until 20 distinct locations are found. After the collection phase, we once again divide each thread into 20 equal interval and inspect the content of each location at the beginning of each interval.

The experimental results clearly show the benefit of our approach. STIQ queries are guaranteed to take $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ CPU time; in practice they never reach the one second mark, and take only a dozen milliseconds on average. In contrast, some TOD queries can take an extremely long time, completely ruining the interactivity of the debugging session^[11].

¹¹ Note that the average query times for TOD are high in great part because of a few extremely long outliers; many queries still execute in a few dozen milliseconds.

Table 5. Cost of capture features as percentage of total capture time

Workload	object ids map	field reads
avroa	9.5%	66%
lusearch	17%	53%
burntest	41%	47%

Table 6. Size of the different indexes as percentage of total index size

Workload	control flow	memory locs	snapshots	strings
avroa	56%	28%	14%	0.6%
lusearch	14%	71%	11%	4%
burntest	1.3%	97%	0.8%	0.7%

Overall, we consider our approach successful. Capture overhead, indexing times and trace sizes are all significantly better than TOD. In addition, STIQ really shines at query processing, always guaranteeing interactive-compatible response times. We are not aware of any system that gives such strong guarantees on query process times.

6.2 Cost of Individual Features

This section gives a detailed accounting of the cost of the different features of STIQ for both capture and indexing. This is useful to pinpoint optimization targets.

Table 5 shows the cost of two important features used at capture time. As mentioned in Sect. 3.1, we must resort to a global map to store the ids of the instances of certain problematic classes. This has a non-negligible cost, that could be avoided if the JVM was modified to allow additional fields to be added to the `Object` class. The non-determinism of memory caused by thread scheduling requires the capture of the values of each memory read. This represents about half the capture time.

Table 6 show how the index size is distributed among the different indexes¹². The distribution varies widely from a workload to another, but it is worthwhile to note that our lightweight snapshots use comparatively very little space.

7 Related Work

We now discuss related work in the areas of back-in-time debugging, deterministic replay, and analysis of captured execution traces.

Back-in-time debugging. TOD [12,13] is our previous attempt at a scalable disk-based back-in-time debugger for Java. It uses a specialized distributed

¹² The strings index stores the values of the strings used in the program. As it is not directly used for queries and has very limited impact in general, we did not mention it elsewhere in this paper.

database to speed up indexing and querying. It is based on exhaustive trace capture and exhaustive indexing of events. As a consequence, its runtime overhead is higher than STIQ (up to 176x vs. up to 30x), and it is very resource hungry (traces are up to 13x larger than STIQ, and indexes up to 177x larger). Moreover, many queries in TOD involve a conjunction on several indexes, requiring a linear scan that can take a long time in some cases (more than a minute). In contrast, our system guarantees $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ CPU time for all queries, in practice not exceeding a few hundred milliseconds.

Amber/Chronicle¹³ by Robert O’Callahan is a back-in-time debugger for native Linux programs that is designed to deal with large execution traces. As TOD, it relies on exhaustive trace capture, and it creates an on-disk index of the execution trace. It performs compression of both trace and index data. It is interesting to note that for indexing memory accesses it uses the principle of *spatial locality*: contiguous instructions that access contiguous memory locations produce a single event. However it does not create an individual index for each memory location, and thus suffers from the same limitation as TOD: it is possible that a large number of entries have to be scanned before finding the correct one. The runtime overhead of trace capture (300x) is also much higher than what we achieve with STIQ.

The Omniscient Debugger [8] and Unstuck [5] are tools for Java and Smalltalk respectively that store the execution trace in RAM, in the same process as the debugged program. Because the amount of available storage is limited, they resort to discarding the oldest events to make room for the new ones. Lienhard *et al.* [11] discard the events that relate to objects that have been garbage collected. In both cases, discarding events can limit the usefulness of the approach, as bugs can have occurred much before the symptoms appear, or in the context of objects that are no longer in use.

The Whyline [6] is a debugging system for Java that provides richer queries than most back-in-time debuggers: it lets the user select questions about why some behavior *did* or *did not* occur. These questions are automatically generated based on a combination of static and dynamic analysis, and can deal not only with the internal state of the program (memory locations, control flow), but also with its textual and graphical output, down to individual pixels. Although the Whyline can analyze relatively large execution traces (*e.g.* 35 million events), its scalability is limited by the fact that the analysis is performed in memory. ZStep [9] is an early back-in-time debugger for Lisp that does not claim great scalability, but instead explores the user interaction aspect of back-in-time debugging. It can also relate graphical output to the event that produced it.

Deterministic replay. Flashback [16] and Jockey [15] are deterministic replay systems for native Linux programs. Flashback relies on a modified kernel while

¹³ Although there are no formal publications about this open-source tool, Amber/Chronicle is a serious endeavor that has been successfully used to debug the Firefox web browser. Information can be found on this page: http://weblogs.mozillazine.org/roc/archives/2006/12/more_about_ambe.html.

Jockey relies on program instrumentation. They both take periodic snapshots of the state of the debugged process and record the interactions between the program and its environment. Snapshots are based on a fork of the process and take advantage of the copy-on-write mechanism of the kernel to avoid having to explicitly copy the entire address space. However, the fact that snapshots have to stay in memory make it necessary to discard older ones. Both systems have a runtime overhead lower than ours (2x-4x for Flashback, up to 30% for Jockey), but they do not properly handle multithreaded programs. Nirvana [11] is a deterministic replay system for native programs that properly supports multithreaded programs. Like our own system, it records the results of memory reads to account for scheduling-induced non determinism. Its runtime overhead is between 5x and 17x, which is slightly better than what we achieve with STIQ.

DejaVu [3] is a deterministic replay system for Java based on modifications of the JVM. It supports multithreaded programs and has a rather low runtime overhead (usually less than 100%), but the JVM used does not have a JIT compiler and thus only runs in interpreted mode, which has very different performance characteristics compared to production JVMs.

Retrace [19] is a deterministic replay system for uniprocessor VMWare virtual machines. It has an extremely low runtime overhead (around 5%) and produces very compact traces. Such a low runtime overhead is possible because the recorded system is the entire (virtual) machine, and therefore the amount of interaction with the environment is limited to mostly IO operations; in particular, thread scheduling and the associated non-determinism on memory locations need not be captured, as the scheduling itself is a deterministic part of the recorded system.

Capture and analysis of execution traces. Capture of execution trace for automatic offline analysis is a well studied topic. Zhang *et al.* [20] present several lossless compression techniques used to record whole execution traces of native programs. These compression algorithms support direct navigation in the compressed traces. Tallam *et al.* [17] show that it is possible to extend control flow traces to indirectly capture runtime data dependencies. Xin *et al.* [18] present a technique to efficiently capture control flow at a level of granularity finer than procedure calls, and provide a numbering scheme of executed statements useful to correlate several executions of the same program. Using the above techniques, relatively complex queries (*e.g.* calculating dynamic slices or matching instruction flows in different versions of the same program) can be executed in seconds or minutes instead of the hours or days it would take using a naive approach. In contrast, with our system, simple queries specific to the typical tasks of back-in-time debugging can be executed in at most a few hundred milliseconds instead of the seconds or minutes it would take without indexing.

8 Conclusion

This paper presented STIQ, a scalable back-in-time debugging approach based on summarized execution trace indexing and querying that favorably compares

with previous approaches on three essential levels: trace capture overhead, indexing speed and query response time. In particular, it leverages deterministic replay for a lower runtime overhead, and indexes only summarized information about bounded-size execution blocks for fast indexing and querying. Importantly, it guarantees that all queries only require $\mathcal{O}(\log n)$ disk accesses and $\mathcal{O}(1)$ CPU time; in practice they never reach the one second mark, and take only a dozen milliseconds on average. Such efficient querying is key to providing an interactive debugging experience; we are not aware of any back-in-time debugging system that provides such strong guarantees.

In this paper we only presented the core queries of back-in-time debuggers (stepping, memory inspection and causality links). However our indexing scheme could easily support other useful queries, such as finding the events that occur on a particular source code line, or the history of objects beyond the history of their individual fields (*e.g.* when objects are passed around as method arguments).

An interesting property of our approach is that the indexing and querying scheme is independent from the technique used for trace capture and replay. The only requirement is that it must be able to obtain (*a*) memory write and method entry/exit events for index construction (in this paper these are obtained through an initial replay that generates a semi-exhaustive trace), and (*b*) exhaustive event lists of arbitrary execution blocks for processing queries (in this paper this is achieved by taking lightweight snapshots that permit to replay such blocks). Although this work provides both the capture and the indexing mechanism, we feel that the capture is still too slow to be really practical. It is our hope that this work will encourage the building of improved capture mechanisms that can be plugged into our indexing system so as to obtain a practical back-in-time debugger. It would be particularly interesting to assess how an extremely efficient capture system such as Retrace [19] could be used for this purpose.

References

1. Bhansali, S., Chen, W.-K., de Jong, S., Edwards, A., Murray, R., Drinić, M., Mihočka, D., Chau, J.: Framework for instruction-level tracing and analysis of program executions. In: VEE 2006: Proceedings of the second international conference on Virtual execution environments, pp. 154–163. ACM Press, New York (2006)
2. Blackburn, S.M., Garner, R., Hoffman, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The DaCapo benchmarks: Java benchmarking development and analysis. In: OOPSLA 2006: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 169–190. ACM Press, New York (2006)
3. Choi, J.-D., Srinivasan, H.: Deterministic replay of Java multithreaded applications. In: SPDT 1998: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 48–59. ACM Press, New York (1988)
4. Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory 21(2), 194–203 (1975)

5. Hofer, C., Denker, M., Ducasse, S.: Design and implementation of a backward-in-time debugger. In: Proceedings of NODE 2006. Lecture Notes in Informatics, vol. P-88, pp. 17–32. Gesellschaft für Informatik, GI (2006)
6. Ko, A.J., Myers, B.A.: Debugging reinvented: Asking and answering why and why not questions about program behavior. In: ICSE 2008: Proceedings of the International Conference on Software Engineering, pp. 301–310 (2008)
7. Larus, J.R.: Whole program paths. In: PLDI 1999: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation, pp. 259–269. ACM Press, New York (1999)
8. Lewis, B.: Debugging backwards in time. In: Ronsse, M., De Bosschere, K. (eds.) Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), Ghent, Belgium, vol. cs.SE/0310016 (2003)
9. Lieberman, H., Fry, C.: ZStep 95: A reversible, animated source code stepper. In: Stasko, J., Domingue, J., Brown, M.H., Price, B.A. (eds.) Software Visualization — Programming as a Multimedia Experience, pp. 277–292. The MIT Press, Cambridge (1998)
10. Lienhard, A., Fierz, J., Nierstrasz, O.: Flow-centric, back-in-time debugging. In: Oriol, M., Meyer, B. (eds.) TOOLS EUROPE 2009. Lecture Notes in Business Information Processing, vol. 33, pp. 272–288. Springer, Heidelberg (2009)
11. Lienhard, A., Girba, T., Wang, J.: Practical Object-Oriented Back-in-Time Debugging. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 592–615. Springer, Heidelberg (2008)
12. Pothier, G., Tanter, É.: Back to the future: Omniscient debugging. *IEEE Software* 26(6), 78–95 (2009)
13. Pothier, G., Tanter, É., Piquer, J.: Scalable omniscient debugging. In: Proceedings of the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2007), pp. 535–552. ACM Press, New York (2007); *ACM SIGPLAN Notices*, 42(10)
14. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA (2010)
15. Saito, Y.: Jockey: a user-space library for record-replay debugging. In: Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging (AADEBUG 2005), pp. 69–76. ACM Press, New York (2005)
16. Srinivasan, S.M., Kandula, S., Andrews, C.R., Zhou, Y.: Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In: ATEC 2004: Proceedings of the Annual Conference on USENIX Annual Technical Conference, pp. 3–3. USENIX Association, Berkeley (2004)
17. Tallam, S., Gupta, R., Zhang, X.: Extended whole program paths. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 17–26 (2005)
18. Xin, B., Sumner, W.N., Zhang, X.: Efficient program execution indexing. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI, pp. 238–248. ACM, New York (2008)
19. Xu, M., Malyugin, V., Sheldon, J., Venkitachalam, G., Weissman, B., Inc, V.: Re-trace: Collecting execution trace with virtual machine deterministic replay. In: Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS (2007)
20. Zhang, X., Gupta, R.: Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.* 2(3), 301–334 (2005)

Interprocedural Exception Analysis for C++

Prakash Prabhu^{1,2}, Naoto Maeda^{1,3}, Gogul Balakrishnan¹, Franjo Ivančić¹,
and Aarti Gupta¹

¹ NEC Laboratories America, 4 Independence Way, Suite 200, Princeton, NJ 08540

² Princeton University, Department of Computer Science, Princeton, NJ 08540

³ NEC Corporation, Kanagawa 211-8666, Japan

Abstract. C++ Exceptions provide a useful way for dealing with abnormal program behavior, but often lead to irregular interprocedural control flow that complicates compiler optimizations and static analysis. In this paper, we present an interprocedural exception analysis and transformation framework for C++ that captures the control-flow induced by exceptions and transforms it into an exception-free program that is amenable for precise static analysis. Control-flow induced by exceptions is captured in a modular interprocedural exception control-flow graph (IECFG). The IECFG is further refined using a novel interprocedural dataflow analysis algorithm based on a compact representation for a set of types called the Signed-TypeSet domain. The results of the interprocedural analysis are used by a lowering transformation to generate an exception-free C++ program. The lowering transformations do not affect the precision and accuracy of any subsequent program analysis. Our framework handles all the features of synchronous C++ exception handling and all exception sub-typing rules from the C++0x standard. We demonstrate two applications of our framework: (a) automatic inference of exception specifications for C++ functions for documentation, and (b) checking the “no-throw” and “no-leak” exception-safety properties.

1 Introduction

Exceptions are an important error handling aspect of many programming languages, especially object-oriented languages such as C++ and Java. Exceptions are often used to indicate unusual error conditions during the execution of an application (resource exhaustion, for instance) and provide a way to transfer control to special-purpose exception handling code. The exception handling code deals with the unusual circumstance and either terminates the program or returns control to the non-exceptional part of the program, if possible. Therefore, exceptions introduce additional, and often complex, interprocedural control flow into the program, in addition to the standard non-exceptional control flow.

The interprocedural control flow introduced by exceptions necessitate global reasoning over whole program scope, which naturally increases the potential for bugs. Stroustrup developed the notion of exception safety guarantees for components [18]. Informally, exception safety means that a component exhibits *reasonable behavior* when an exception is raised. The term “reasonable” includes

all the usual expectations for error-handling: resources should not be leaked, and that the program should remain in a well-defined state so that execution can continue. Stroustrup introduced various degrees of exception safety guarantees that can be expected from components:

- **No leak guarantee:** If an exception is raised, no resources such as memory are leaked.
- **Basic guarantee:** In addition to the no leak guarantee, the basic invariants of components (for example, properties that preserve data structure integrity) are maintained.
- **Strong guarantee:** In addition to the basic guarantee, this requires that an operation either succeeds or has no effect, if an exception is raised.
- **No throw guarantee:** In addition to the basic guarantee, this requires that an operation is guaranteed not to raise an exception.

However, it is very difficult to ensure such exception-safety properties, because developers may overlook exceptional control-flow hidden behind multiple levels of abstraction. For instance, in a code block containing local objects as well as exceptions, programmers have to reason about non-local returns induced by exceptions, and at the same time understand the effects of the implicit calls to the destructors of local objects along the exception path correctly. Unlike Java, all C++ exceptions are unchecked, and library developers are not required to annotate interfaces with exception specifications. Furthermore, dynamic exception specifications (anything other than `noexcept` specification) are deprecated in the latest C++0x draft standard [17]. Consequently, developers increasingly rely on documentation to discern throwable exceptions from a function interface (more so in the absence of source code), which makes it hard to reason about programs that use library functions that throw exceptions. Therefore, a tool that automatically models the behavior of exceptions precisely would be useful.

Existing Approaches to C++ Exceptions. Program analysis techniques, both static and dynamic, are often applied in the context of program optimization, automatic parallelization, program verification, and bug finding. These techniques rely heavily on both intraprocedural and interprocedural control flow graph information, which are utilized to compute relevant information as needed (e.g., dependence analysis or program slicing). However, existing compiler frameworks for C++ (for example `g++`, `clang/LLVM` [10]) do not build precise models for exceptions. Specifically, they only analyze exceptional control flow within a locally declared *trycatch* statement, and do not perform either an intraprocedural or interprocedural analysis. Therefore, they make conservative assumptions about interprocedural control flow, which causes their models to include paths between *throw* statements and *catch* blocks that are infeasible at runtime. An alternative approach is to use such frameworks to generate a semantically equivalent C program from the given C++ program and use the lowered C code for further analysis. However, the code generated by these tools use custom data structures and involves calls into opaque runtimes, which need to be modeled conservatively in static analysis algorithms, resulting in further loss in precision.

Our Approach. In this paper, we present an interprocedural exception analysis and transformation framework for C++ that (1) captures the control-flow induced by exceptions precisely, and (2) transforms the given C++ into an exception-free program that is amenable for precise static analysis. We summarize our contributions below:

- We propose a *modular* abstraction for capturing the interprocedural control flow induced by exceptions in C++, called the interprocedural exception control flow graph (IECFG). The IECFG is constructed through a sequence of steps, with each step refining it. The modular design of IECFG is motivated by the need to model implicit calls to destructors during stack unwinding, when an exception is thrown. The modularity of IECFG is also important in practice, for permitting re-use in presence of separate compilation units.
- We design and implement an interprocedural exception analysis algorithm to model the set of C++ exceptions that reach *catch* statements in the program using the *Signed-TypeSet* domain, which represents a set of program types compactly. Our analysis is formulated in conjunction with the construction of the IECFG. A unique feature of our framework is the capability to safely terminate the IECFG construction at certain well-defined points during interprocedural propagation, thereby, allowing clients, such as optimizing compilers or program analysis, to trade-off speed over precision.
- We present a lowering algorithm that uses the results of our exception analysis to generate an exception-free C++ program. Unlike standard compilers, our algorithm does not use non-local jumps or calls into any opaque C++ runtime systems. Absence of an external runtime and non-local jumps enables existing static analyses and verification tools to work soundly over C++ programs with exceptions, without needing to model them explicitly within their framework. While the IECFG construction is modular in the sense of allowing separate compilation units, the lowering algorithm to generate exception-free code is not modular. It requires a global view of all source code under analysis so that all known possible targets of virtual function calls can be determined.
- We present the results of using our interprocedural exception analysis and transformation framework on a set of C++ programs. We compute the exception specifications for functions and check the related “no throw” guarantee. We also check the “no leak” exception-safety property.

Example. Consider the C++ program shown in Fig. 1. The program has three functions, of which `get()` allocates a `File` object and attempts to read a line from `File` by calling `readLine()`. If the file does not exist, `readLine()` throws an `IOException` that is handled in the `get()` function. Otherwise, a call is made to `read()` which throws an `EOFException` if the end of the file is reached, which is handled in `readLine()`. An exception modeling framework has to abstract the interprocedural control flow due to exceptions correctly, and also take into account the implicit calls made to destructors during stack unwinding, when an exception propagates out of a function (e.g., destruction of `str` in `read()` when `EOFException` is thrown).

```
string File::read() {
    string str(__line__);
    if (EOF)
        throw EOFException();
    return str;
}
```

```
class EOFException { ... };
class IOException { ... };
```

```
void get () {
    string s;
    try {
        File *file = new File("l.txt");
        file->readLine();
        delete file;
    }
    catch (IOException& ie) {
        cout<< "IO-Failure";
    }
    return;
}
```

```
string File::readLine() {
    string s;
    try {
        if (invalidFile)
            throw IOException();
        s = read();
        return s;
    }
    catch (EOFException& e) {
        return string("");
    }
}
```

Fig. 1. Running Example

There are two bugs worth noting in this example, both of which have to do with exceptions: (1) violation of “no leak” guarantee, the `file` object gets leaked along the exception path from `readLine()` to the `catch` block in `get()`, and (2) violation of “no throw” guarantee, a potential `std::bad_alloc` exception thrown by `new` is not caught in `get()`. Our exception analysis and transformation framework enables checking these properties easily. ■

Comparison with Java Exception Analysis. Several analysis approaches for modeling Java exceptions have been proposed in the recent past. Most approaches [3, 8, 9] compute an interprocedural exception control flow graph as we do. There have been some attempts to analyze the “no leak” exception-safety guarantee for Java programs also [11]. However, there are a number of major differences between exception handling in Java and C++, which require different design decisions in comparison to Java-based exception analysis techniques:

1. In C++, when an exception propagates out of a function, destructors are invoked on all stack-allocated objects between the occurrence of the exception and the catch handler in a process called *stack unwinding*. Stack unwinding in C++ is a major difference compared to Java, and raises various performance issues, along with complicating the modeling of exceptional control-flow.
2. C++ destructors can call functions which throw exceptions internally, leading to a scenario where multiple exceptions are live during stack unwinding. Unlike implicitly invoked destructors, Java provides “finalizers”, that are invoked non-deterministically by the garbage collector. Although the use of “finally” blocks in Java can result in multiple live exceptions, these blocks are created and controlled explicitly by the programmer, and therefore, multiple live exceptions in Java are apparent from the code itself.
3. The exception subtyping rules for Java are limited to only parent-child relationships within the class hierarchy. Besides, all exception classes trace their

lineage to a single ancestor, the `Exception` class. In contrast, C++ exception subtyping rules are richer and include those concerning multiple inheritance, reference types, pointers, and few other explicit type-conversion rules among functions as well as arrays.

4. The exception specification and checking mechanism in Java is much stronger than in C++. In particular, Java has a “checked exception” category of exceptions, which explicitly requires programmers to either catch exceptions thrown within a function, or declare them as part of the interface. C++ has no concept of checked exceptions, and the dynamic exception specifications are deprecated in the latest C++0x standards draft. Exception specifications in C++ may not even be accurate, which results in a call to `std::unexpected()` function, which may be redefined by an application.
5. C++ provides an exception probing API while Java does not. It provides a means to conditionally execute code depending on whether there is an outstanding live exception by calling `std::uncaught_exception()`. This can be used to decide whether or not to throw exceptions out of a destructor. C++ also allows users to specify abnormal exception termination behavior by providing custom handlers for `std::terminate()` or `std::unexpected()`.
6. Java exceptions are handled based on runtime types, whereas in C++ static type information is used to decide which catch handler is invoked. Therefore, pointer analysis is required in Java to improve the accuracy of matching throw statements with catch blocks. For C++, we can avoid a heavy duty pointer analysis for exceptions. (However, call graph construction in C++ can be improved with the results of a pointer analysis on function pointers and virtual function calls.)

2 Preliminaries

We first describe the abstract syntax of a simplified intermediate language (IL) for C++ used within our framework. The language is based on CIL [14], with additional constructs for object-oriented features. Fig. 2 shows the subset of the actual IL that is relevant for the exception analysis and transformation framework. Types within our IL include the primitive ones (*int*, *float*, *void*) as well as user defined classes (*cl*), derived types (pointer, reference and array), function types, and can additionally be qualified (*const*, *volatile*, *restrict*). Each class type can inherit from a set of classes, and has a set of fields and member functions, some of which may be *virtual*. Visibility of the class members and the inherited classes is controlled by an access specifier.

A program is a set of globals. A global is either a type or a function. A function has a signature and a body, which is a block of statements. Statements include instructions, regular control flow statements (*loop*, *if*, *trycatch*), irregular control flow statements causing either local (*goto*, *break*, *continue*) or global (*throw*, *return*) alterations. An instruction is one of the following: an assignment, an allocation operator, a deallocation operator, a global function call, or a member function call. Expressions could involve binary operators, unary operators, pointer dereferences or indirections, reference indirections, and cast operations.

Constant	c	\in	$Constant$
Identifiers	id	\in	$Identifier$
Labels	l	\in	$Label$
Access	a	$::=$	$private \mid protected \mid public$
Qualifier	cv	$::=$	$const \mid volatile \mid restrict$
Class	cl	$::=$	$class \ id : \ \overline{a \ t} \ \{ \overline{a \ t \ f i; a \ virtual? \ m} \}$
Type	t	$::=$	$id \mid t * \mid t \ \& \mid t [e] \mid t \rightarrow t \mid void \mid int \mid float \mid \overline{cv} \ t$
Variable	v	$::=$	id
Lvalue	lv	$::=$	$lh \ e$
Lhost	lh	$::=$	$v \mid * \ e$
Program	p	$::=$	\overline{g}
Global	g	$::=$	$t \ id \mid f$
Function	f	$::=$	$t \ id \ (\overline{t \ id}) = b$
Block	b	$::=$	$\{ \overline{s} \}$
Statement	s	$::=$	$i \mid return \ e? \mid goto \ l \mid break \mid continue \mid$ $if \ e \ b_1 \ b_2 \mid loop \ b \mid throw \ e? \mid trycatch \ b \ \overline{h}$
Handler	h	$::=$	$(t \ v) \ b \mid (\dots) \ b$
Instruction	i	$::=$	$call \ id \ e_f \ \overline{e} \mid mbrcall \ id \ e_{this} \ e_f \ \overline{e} \mid e := e \mid v := new \ e \mid delete \ e$
Cast	$cast$	$::=$	$staticcast \mid dyncast \mid constcast \mid reinterpretcast$
Expression	e	$::=$	$c \mid lv \mid unop \ e \mid e \ binop \ e \mid cast \ t \ e \mid \&lv \mid lv\&$

Fig. 2. Abstract Syntax of the Simplified IL for C++

C++ Exceptions. C++ exceptions are *synchronous*. Asynchronous exceptions, which in Java are raised due to internal errors in the virtual machine, are categorized as program errors in C++ and are not handled by the exception constructs of C++. Synchronous exceptions, in contrast, are expected to be handled by the programmer and are only thrown by certain statements in the program, such as (a) *throw* statement, which throws a fresh exception or rethrows a caught exception, (b) function call, which transitively throws exceptions uncaught within its body or its callees, (c) *new* operator, which can throw a `std::bad_alloc` exception, and (d) *dynamic_cast*, which can throw a `std::bad_cast` exception.

Exception Handling and Subtyping rules for C++. Exceptions in C++ are caught using exception handlers, defined as part of the *trycatch* statement. Each *trycatch* statement has a single *try* block followed by a sequence of exception (*catch*) handlers. An exception object thrown from within the *try* block is caught by the first handler whose declared exception type matches the thrown exception type according to the C++ exception subtyping rules. If no match is found for a thrown object amongst the handlers, control flows either to an enclosing *trycatch* statement or out of the function to the caller.

The exception subtyping rules for C++ as defined in the final C++0x draft standard [17] are shown in Fig. 3. The type of a thrown exception is given by t_T and the type declared in the exception handler is given by t_C in each rule. A handler is a match for an exception, if any of the following conditions hold:

- The handler’s declared type is the same as the type of the exception, even when ignoring the const-volatile qualifiers (Rules **EQ** and **CVQUAL**).
- The handler’s declared type is an unambiguous public base class of the exception type. Arrays are treated as pointers and functions returning a type are treated as pointers to function returning the same type (Rules **SUBCL**, **ARR**, **FPTR** and **CVQUAL**).
- The handler’s declared type is a reference to the exception type (Rule **REF**).

$$\begin{array}{c}
\frac{t_T = t \quad t_C = t}{t_T \leq t_C} \text{ [EQ]} \\
\frac{t_T = t_1 \quad t_C = t_2 \quad t_1 \in \text{sub}(t_2)}{t_T \leq t_C} \text{ [SUBCL]} \\
\frac{t_T = t_1 \quad t_C = (- \rightarrow t_2) \quad t_1 \leq (- \rightarrow t_2)*}{t_T \leq t_C} \text{ [FPTR]} \\
\frac{t_T = t_1 * \quad t_C = t_2 * \quad t_1 * \leq_{\text{conv}} t_2 *}{t_T \leq t_C} \text{ [PTR]} \\
\frac{t_T = t_1 * \quad t_C = \text{void} *}{t_T \leq t_C} \text{ [VOID]} \\
\frac{t_T = cv t \quad t_C = t}{t_T \leq t_C} \text{ [CVQUAL]} \\
\frac{t_T = t_1 \quad t_C = t_2[] \quad t_1 \leq t_2 *}{t_T \leq t_C} \text{ [ARR]} \\
\frac{t_T = t \quad t_C = t \&}{t_T \leq t_C} \text{ [REF]} \\
\frac{t_T = \text{std} : : \text{nullptr}_t \quad t_C = t *}{t_T \leq t_C} \text{ [NULLPTR]}
\end{array}$$

Fig. 3. Exception Subtyping Rules

- The handler’s declared type is a pointer into which the exception type, which also is a pointer, can be converted using C++ pointer conversion rules (Rule **PTR**).
- The two remaining rules concern generic pointers modeled by `void *` and `std::nullptr_t` [17] (Rules **NULLPTR** and **VOID**).

3 Signed-TypeSet Domain

In this section, we present a novel abstract domain for compactly representing a set of program types, which we call the *Signed-TypeSet* domain.

Definition 1. *The Signed-TypeSet domain Γ is defined as: $\Gamma = \{(s, T_{prog}) \mid s \in \{+, -\}, T_{prog} \subseteq \{t \mid t \text{ is a program exception type}\}$*

The semantics of a positive set of exception types is the standard one, while a negative set of exception types represents “every exception type other than those in the set”. For instance $(+, \{IOException\})$ represents the *IOException* program type alone, while $(-, \{IOException, EOFException\})$ represents any exception type other than *IOException* and *EOFException*. Exceptions thrown by unknown library calls are modeled concisely as $(-, \{\})$. For external library calls, a special unknown exception type $t_{unknown}$ is introduced explicitly only at the point when the lowering transformation is to be done.

We use a signed domain in our framework, rather than a domain of only positive set of program types to make the IECFG computation modular. Its use is especially beneficial in the presence of unknown library calls and deprecated use of dynamic exception specifications in C++. A negated set of types succinctly captures the unknown exceptions that could potentially be thrown by opaque library calls that are not caught. We would also like to incrementally integrate the results of exception analysis from separately compiled functions whenever available, while at the same time maintaining a safely analyzable exception result at all intermediate points. Therefore, our exception dataflow analysis begins with an over-approximation of the set of all exception types that could be raised by a throwable statement, and refines the set via interprocedural propagation. This

<hr/> <p style="text-align: center;">Algorithm 1: union (\cup_Γ)</p> <hr/> <p>Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$ Output: $\tau_c \in \Gamma$</p> <pre> 1 case $\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)$ 2 $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \in T_b\}$; 3 case $\tau_a = (-, T_a) \wedge \tau_b = (+, T_b)$ 4 $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \notin T_b\}$; 5 case $\tau_a = (+, T_a) \wedge \tau_b = (-, T_b)$ 6 $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \notin T_a\}$; 7 case $\tau_a = (+, T_a) \wedge \tau_b = (+, T_b)$ 8 $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_a \vee t \in T_b\}$; 9</pre> <hr/>	<hr/> <p style="text-align: center;">Algorithm 2: intersection (\cap_Γ)</p> <hr/> <p>Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$ Output: $\tau_c \in \Gamma$</p> <pre> 1 case $\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)$ 2 $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \vee t \in T_b\}$; 3 case $\tau_a = (-, T_a) \wedge \tau_b = (+, T_b)$ 4 $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \notin T_a\}$; 5 case $\tau_a = (+, T_a) \wedge \tau_b = (-, T_b)$ 6 $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \notin T_b\}$; 7 case $\tau_a = (+, T_a) \wedge \tau_b = (+, T_b)$ 8 $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \in T_a\}$; 9</pre> <hr/>
<hr/> <p style="text-align: center;">Algorithm 3: set difference ($-\Gamma$)</p> <hr/> <p>Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$ Output: $\tau_c \in \Gamma$</p> <pre> 1 case $\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)$ 2 $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \notin T_a\}$; 3 case $\tau_a = (-, T_a) \wedge \tau_b = (+, T_b)$ 4 $\tau_c = (-, T_c)$ where $T_c = \{t \mid t \in T_a \vee t \in T_b\}$; 5 case $\tau_a = (+, T_a) \wedge \tau_b = (-, T_b)$ 6 $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_b \wedge t \in T_a\}$; 7 case $\tau_a = (+, T_a) \wedge \tau_b = (+, T_b)$ 8 $\tau_c = (+, T_c)$ where $T_c = \{t \mid t \in T_a \wedge t \notin T_b\}$; 9</pre> <hr/>	<hr/> <p style="text-align: center;">Algorithm 4: equals ($=_\Gamma$)</p> <hr/> <p>Input: $\tau_a \in \Gamma, \tau_b \in \Gamma$ Output: bool</p> <pre> 1 case $(\tau_a = (-, T_a) \wedge \tau_b = (-, T_b)) \vee$ $(\tau_a = (+, T_a) \wedge \tau_b = (+, T_b))$ 2 if $(T_a \subseteq T_b) \wedge (T_b \subseteq T_a)$ then 3 true 4 end 5 else 6 false 7 end 8 case default 9 false 10</pre> <hr/>

Fig. 4. Operations on the Signed-TypeSet domain

deliberate design decision allows clients to terminate the analysis at any point during the analysis and safely use the refined IECFG structure at that point for other analyses.

We define set operations on Γ , that are as efficient as the set operations on normal sets. These operations (shown in Fig. 4) mimic the normal set union, intersection, difference, and equality operations. Given two elements τ_a and τ_b from Γ , these operations result in another element τ_c in Γ . All the operations perform a case analysis on the signs of the two elements, and perform normal set operations on the constituent set of program types. The operations are fairly straightforward, and in their full generality need to take into account the exception subtyping rules described earlier. For the sake of conceptual simplicity, we assume that the set of exception types arising at the catch blocks have already been expanded to contain all possible “exception subtypes” in the program, before doing the specific set operations. However, in our implementation, we perform these operations without doing full expansion and correctly account for exception subtypes on demand.

4 Intraprocedural Exception Control Flow Graph

An intraprocedural exception control flow graph is defined as follows:

Definition 2. *An intraprocedural exception control flow graph (IECFG) for a function f , denoted by G_{intra_f} is a tuple $\langle N, E_{reg}, E_{except}, E_{excepts}, E_c, n_s, n_e, n_{except} \rangle$ with Signed-TypeSet domain Γ , where*

- N is the set of nodes in the graph, consisting of the following distinct subsets:

$$N = N_{reg} \cup N_c \cup N_{cret} \cup N_{ecret} \cup N_{throw} \cup N_{catch} \cup \{n_s, n_e, n_{excepe}\}$$
 where
 - N_{reg} is the set of regular nodes.
 - N_c is the set of call nodes.
 - N_{cret} is the set of call-return nodes.
 - N_{ecret} is the set of exceptional-call-return nodes.
 - N_{throw} is the set of throw, new, or dynamic_cast nodes.
 - N_{catch} is the set of header nodes of catch blocks.
 - n_s, n_e, n_{excepe} are unique start, exit and exceptional-exit nodes.
- E_{reg} is the set of regular control flow edges: $E_{reg} \subseteq (N_{reg} \cup N_{cret} \cup N_{catch}) \times N$
- E_{excep} is the set of intraprocedural exception control flow edges:

$$E_{excep} \subseteq ((N_{ecret} \cup N_{throw}) \times (N_{catch} \cup \{n_{excepe}\}) \times \Gamma)$$
- E_{exceps} is the set of exception-call-summary edges:

$$E_{exceps} \subseteq (N_c \times N_{ecret} \times \Gamma)$$
- E_c is the set of normal call-summary edges:

$$E_c \subseteq N_c \times N_{cret}$$

Example (ECFG structure). The ECFGs for the functions in our running example are shown in Fig. 5. Consider the ECFG for the `get()` function. In addition to the start (*s-get*) and exit (*e-get*) nodes present in regular CFGs, the ECFG has a new exceptional-exit (*exe-get*) node. Control flows through an exceptional-exit node, every time an exception propagates out of a function. Each call instruction is represented by three nodes: in addition to the call node (*c-readLine*) and call-return node (*cr-readLine*) present in regular CFGs, the ECFG has a new exceptional-call-return node (*ecr-readLine*) through which control flows when the callee terminates with an exception. The ECFG has two additional exception-related nodes: throw node, one for every potential throwing statement, and catch-header node, one for every catch block in the code.

There are three kinds of edges in a ECFG: (a) normal control-flow edge (solid lines) as in any CFG, (b) normal call-summary edges (long-dashed lines) between call and call-return nodes, and (c) exception edges (short-dashed lines) which can either be a summary edge (between call and exceptional-call-return nodes) or a normal exception edge (for intraprocedural exceptions). Every exception edge is annotated with an element from the Signed-TypeSet domain Γ . The exception annotations represent the dataflow facts used in our interprocedural exception analysis, as described in Sect. 5. ■

ECFG Construction. The algorithm to construct the ECFG of a program performs a post-order traversal on the abstract syntax tree (AST) of the C++ code, creating a set of ECFG nodes and edges as it visits each AST node. Each *visit* method returns a triple $\langle N_b, N_e, N_{unres} \rangle$, where N_b and N_e represent the set of nodes corresponding to start and normal exit of the ECFG “region” corresponding to the current AST node. N_{unres} represents the set of nodes in an ECFG region that has some unresolved incoming or outgoing edges, which are resolved by an ancestor’s visitor. For instance, a *throw* node that is not enclosed

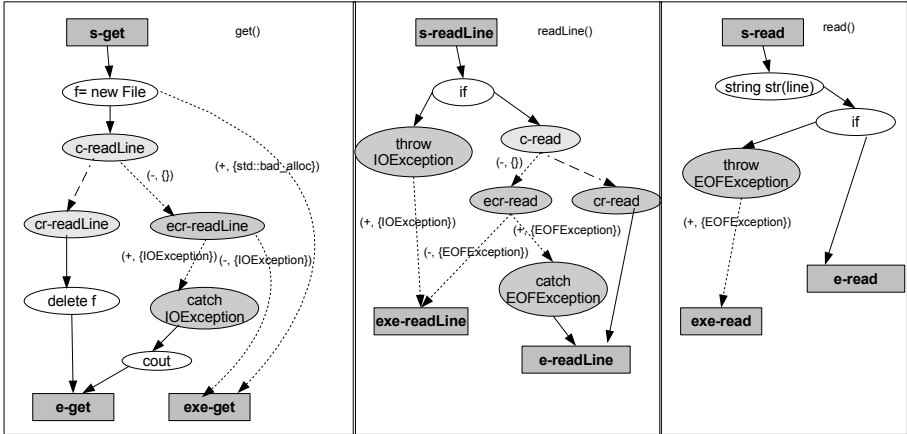


Fig. 5. Intraprocedural Exception Control Flow Graphs for the Program in Fig. 1

within a *trycatch* block is resolved at the root (function declaration), by creating an exception edge to the exceptional-exit node. Alg. 5 shows the visit routine for a *trycatch* statement.

The *VisitTryCatchStmt* routine for *trycatch* statements works as follows: it first constructs the ECFG nodes and edges for the *try* block and all the catch handlers, by visiting them recursively (Lines 1-4). It then divides the exception-related ECFG nodes in the *try* block into two sets: (a) the throw nodes¹, and (b) the exceptional-call-return nodes (Lines 5-7). For the throw nodes, a match is sought in the sequence of handlers by applying the C++ exception subtyping rules described earlier. As soon as the first match is found, an exception edge is created from the *throw* node to the *catch* header node, annotated with the appropriate exception information from the Signed-TypeSet domain (Lines 8-18). For the throw nodes, this information is always a positive set with a singleton exception type, which is the type of the throw expression.

For an exceptional-call-return node n_{ecret} , a map ECR_{Γ} is used during the search for an exception-type match amongst the handlers. ECR_{Γ} maps an exceptional-call-return node to an element from the Signed-TypeSet domain. Initially, ECR_{Γ} holds “all exception types” $((-, \{\}))$, which represents the most conservative assumption as far as possible exceptions thrown from a call are concerned, for all exceptional-call-return nodes. Every time a catch block is encountered and a possible match occurs, $ECR_{\Gamma}(n_{ecret})$ is incrementally updated to hold the “remaining exception types” that could be thrown from this call (using the difference operator $-_{\Gamma}$). The final value of $ECR_{\Gamma}(n_{ecret})$ is used to annotate the edge between the n_{ecret} and the exceptional-exit node of the function. At every match with a catch header node, an exception edge is created from

¹ In C++ programs, `new` and `dynamic_cast` operators may also throw `bad_alloc` and `bad_cast` exceptions, respectively. For sake of clarity, Alg. 5 only considers `throw`. Our implementation deals with `new` and `dynamic_cast` operators properly.

Algorithm 5: *VisitTryCatchStmt*

```

Input:  $s$ : A trycatch statement where  $s = b_1 (h_1, h_2, \dots, h_k)$ 
Output:  $N_b \times N_e \times N_{unres}$ 
1 let  $(N_{b_1}, N_{e_1}, N_{unres_1}) = \text{VisitBlock}(b_1)$ ; // Creates ECFG region for a block
2 foreach  $h_i$  do
3   let  $(N_{b_{h_i}}, N_{e_{h_i}}, N_{unres_{h_i}}) = \text{VisitHandler}(h_i)$  // Creates ECFG region for a handler
4 end
5 let  $S_{throw} = \{n \mid n \in N_{unres_1} \wedge ir(n) = \text{throw}\}$ ;
6 let  $S_{secret} = \{n \mid n \in N_{unres_1} \wedge ir(n) = \text{call} \wedge n \in N_{e_{secret}}\}$ ;
7 let  $S_{rest} = N_{unres_1} - (S_{throw} \cup S_{secret})$ ;
8 foreach  $n_{th} \in S_{throw}$  do
9   for  $i=1$  to  $k$  do
10     $e_{T_{catch}} = (+, \{t\})$  where  $h_i = (t \ v)$   $b$ ;
11     $t_{catch} = t$  where  $h_i = (t \ v)$   $b$ ;
12     $n_{catch_i} = n$  where  $(n \in N_{b_{h_i}}) \wedge (ir(n) = \text{catch})$ ;
13     $\tau_{throw} = (+, \{t'\})$  where  $(ir(n_{th}) = \text{throw } e) \wedge (T_{prog}(e) = t')$ ;
14    if  $t' \leq t_{catch}$  then
15       $E_{excep} = E_{excep} \cup \{(n_{th}, n_{catch_i}, \tau_{throw})\}$ ;
16       $N_{done} = N_{done} \cup \{n_{th}\}$ ;
17      continue: foreach outer loop
18    end
19  end
20 foreach  $n_{ecret} \in S_{secret}$  do
21   for  $i=1$  to  $k$  do
22     $\tau_{catch} = (+, \{t\})$  where  $h_i = (t \ v)$   $b$ ;
23     $\tau_{ecret} = \text{ECR}_\Gamma(n_{ecret})$ ;
24     $\tau_{intersect} = \tau_{ecret} \cap_\Gamma \tau_{catch}$ ;
25     $\tau_{remain} = \tau_{ecret} -_\Gamma \tau_{catch}$ ;
26     $\text{ECR}_\Gamma(n_{ecret}) = \tau_{remain}$ ;
27     $E_{excep} = E_{excep} \cup \{(n_{ecret}, n_{catch_i}, \tau_{intersect})\}$ ;
28   end
29 end
30  $N_{reg} = N_{reg} \cup \{n_{join}\}$ ;
31  $E_{reg} = E_{reg} \cup \{(n, n_{join}) \mid n \in N_{e_1}\} \cup \{(n, n_{join}) \mid \exists i \text{ such that } n \in N_{e_{h_i}}\}$ ;
32  $N_b = N_{b_1}$ ;  $N_e = \{n_{join}\}$ ;  $N_{unres} = (\cup_{1 \leq i \leq k} N_{unres_{h_i}} \cup N_{unres_1}) - N_{done}$ 

```

Fig. 6. *VisitTryCatch* routine for ECFG construction

the exceptional-call-return node to the catch header, annotated with appropriate exception information (Lines 20-28). The *VisitTryCatchStmt* routine creates a header node (n_{try}) and a join (n_{join}) node, in addition to those created by its children (Lines 30-32).

The handling of the exceptional-call-return node illustrates one distinguishing feature of our approach, as compared to other exception analysis algorithms proposed for Java: the dataflow facts are initialized with an over-approximation, which can be refined using the exception information from the catch-header node. Our choice of the Signed-TypeSet domain, which allows a negative set of exception types, not only permits modeling of unknown library calls within the same framework, but also permits a safe termination of our analysis at any point after the construction of the intraprocedural exception flow graphs.

Figs. 7, 8, and 9 show the *Visit* routines for the remaining IL constructs. The *VisitBlock* routine is straightforward, recursively visiting each child statement and then creating edges between corresponding nodes. The *VisitStmt* routine creates ECFG nodes and edges differently based on the type of AST node. For *if* and *loop* statements, it creates ECFG regions with appropriate join and header

Algorithm 6: *VisitBlock*

Input: b : Block where $b = s_1, s_2, \dots, s_n$
Output: $N_b \times N_e \times N_{unres}$

```

1 for  $i = 1$  to  $n$  do
2   |  $\text{let } (N_{b_i}, N_{e_i}, N_{unres_i}) = \text{VisitStmt}(s_i)$ ;
3 end
4  $E_{reg} = E_{reg} \cup \{(n_1, n_2) \mid \forall i. 1 \leq i < n. (n_1 \in N_{e_i} \wedge n_2 \in N_{b_{i+1}})\}$ ;
5  $N_b = N_{b_1}; N_e = N_{e_n}; N_{unres} = \cup_{1 \leq i \leq n} N_{unres_i}$ ;
```

Algorithm 7: *VisitStmt*

Input: s : Statement
Output: $N_b \times N_e \times N_{unres}$

```

1 switch  $\text{typeOf}(s)$  do
2   case  $\text{instr} \in \{\text{call}, \text{mbrcall}\}$ 
3     |  $\text{VisitCallInstr}(i)$ ;
4   case  $\text{instr} \notin \{\text{call}, \text{mbrcall}\}$ 
5     |  $N_{reg} = N_{reg} \cup \{n_i\}; N_b = N_e = \{n_i\}; N_{unres} = \{\}$ 
6   case  $\text{break} \mid \text{continue} \mid \text{goto} \mid \text{return}$ 
7     |  $N_{reg} = N_{reg} \cup \{n_s\}; N_b = N_{unres} = \{n_s\}; N_e = \{\}$ ;
8   case  $\text{if } e \text{ } b_1 \text{ } b_2$ 
9     |  $\text{let } (N_{b_1}, N_{e_1}, N_{unres_1}) = \text{VisitBlock}(b_1)$ ;
10    |  $\text{let } (N_{b_2}, N_{e_2}, N_{unres_2}) = \text{VisitBlock}(b_2)$ ;
11    |  $N_{reg} = N_{reg} \cup \{n_{ife}, n_{join}\}$ ;
12    |  $E_{reg} = E_{reg} \cup \{(n_{ife}, n_{b_1}) \mid n_{b_1} \in N_{b_1}\} \cup \{(n_{ife}, n_{b_2}) \mid n_{b_2} \in N_{b_2}\}$ 
13    |  $\cup \{(n_{e_1}, n_{join}) \mid n_{e_1} \in N_{e_1}\} \cup \{(n_{e_2}, n_{join}) \mid n_{e_2} \in N_{e_2}\}$ ;
14    |  $N_b = \{n_{ife}\}; N_e = \{n_{join}\}; N_{unres} = N_{unres_1} \cup N_{unres_2}$ ;
15   case  $\text{loop } b_l$ 
16     |  $\text{let } (N_{b_l}, N_{e_l}, N_{unres_l}) = \text{VisitBlock}(b_l)$ ;
17     |  $\text{let } S_{continue} = \{n \mid n \in N_{unres_l} \wedge \text{ir}(n) = \text{continue}\}$ ;
18     |  $\text{let } S_{break} = \{n \mid n \in N_{unres_l} \wedge \text{ir}(n) = \text{break}\}$ ;
19     |  $E_{reg} = E_{reg} \cup \{(n_c, n_{lhead}) \mid n_c \in S_{continue}\} \cup \{(n_b, n_{lexit}) \mid n_b \in S_{break}\}$ ;
20     |  $N_{reg} = N_{reg} \cup \{n_{lhead}, n_{lexit}\}$ ;
21     |  $N_b = \{n_{lhead}\}; N_e = \{n_{lexit}\}; N_{unres} = N_{s_1} - (S_{continue} \cup S_{break})$ ;
22   case  $\text{throw} \mid \text{new} \mid \text{dynamic\_cast}$ 
23     |  $\text{VisitThrowingStmt}(s)$ ;
24   case  $\text{trycatch}$ 
25     |  $\text{VisitTryCatchStmt}(s)$ ;
26
27 endsw
```

Fig. 7. *VisitBlock* and *VisitStmt* routines for ECFG Construction

nodes (Lines 8-21), while for some of the non-exception unstructured control flow statements like *break*, *goto*, it creates unresolved nodes (Lines 6-7), which are patched later on by their parents.

The *VisitCallInstr* routine creates three nodes and two edges for a call instruction. The three nodes are a call node n_{c_i} , a call-return node (n_{ecret_i}) and an exceptional-call-return node (n_{ecret_i}). The two edges are a summary edge connecting n_{c_i} with n_{ecret_i} , and an exceptional-summary edge connecting n_{c_i} with n_{ecret_i} . The exceptional-summary edge is annotated with the most approximate element $(-, \{\})$, which represents any exception type. The exception return node n_{ecret_i} is unresolved since its targets are determined by the enclosing *trycatch* statements or by the exceptional-exit-node at function scope.

The *VisitThrowingStmt* routine creates an unresolved node for the *throw* statement, while the *VisitHandler* routine creates a header node corresponding to the *catch* block, and connects it to the nodes created for the statements within the block. The *VisitFunction* routine is the main driver for creating the

Algorithm 8: *VisitCallInstr*

Input: i : A *call/mbrcall* Instruction
Output: $N_b \times N_e \times N_{unres}$

- 1 $N_c = N_c \cup \{n_{c_i}\}$; $N_{cret} = N_{cret} \cup \{n_{cret_i}\}$; $N_{ecret} = N_{ecret} \cup \{n_{ecret_i}\}$;
- 2 **let** $\tau_e = (-, \{\})$;
- 3 $E_c = E_c \cup \{(n_{c_i}, n_{cret_i})\}$; $E_{excep} = E_{excep} \cup \{(n_{c_i}, n_{ecret_i}, \tau_e)\}$;
- 4 $N_b = \{n_{c_i}\}$; $N_e = \{n_{cret_i}\}$; $N_{unres} = \{n_{ecret_i}\}$;

Algorithm 9: *VisitThrowingStmt*

Input: s : A *throw/new/dynamic.cast* statement
Output: $N_b \times N_e \times N_{unres}$

- 1 $N_{throw} = N_{throw} \cup \{n_s\}$;
- 2 $N_b = N_{unres} = \{n_s\}$; $N_e = \{\}$;

Fig. 8. *VisitCallInstr* and *VisitThrowingStmt* routines for ECFG construction

ECFG for a function. Once the ECFG region for the function body is created, this routine connects the *return* nodes to the normal exit nodes and *throw* nodes to the exceptional-exit node. It finally resolves the unmatched exceptional-call-return nodes by connecting them to the exceptional-exit node, annotated with appropriate exception type annotation.

Example. In Fig. 5, the ECFG for `get()` has an exceptional-call-return node for `readLine()`. The algorithm creates an exception edge from this node to a catch header that handles `IOException` exceptions and annotates the edge with $(+, \{IOException\})$. The algorithm then creates an exception edge to the exceptional-exit node of `get()` annotated with $(-, \{IOException\})$, which is meant to read “If `readLine()` throws any exception other than `IOException`, control is transferred to the exceptional exit node of `get()`”. ■

5 Interprocedural Exception Analysis

Once the intraprocedural graphs have been constructed, they are connected together to form an interprocedural exception control flow graph, which is defined as follows:

Definition 3 An interprocedural exception graph (IECFG) is defined by the tuple $I_G = \langle N_s, N_e, N_{excepe}, G_{inter} \rangle$, where N_s, N_e, N_{excepe} are the set of start, exit and exceptional-exit nodes of the constituent ECFGs, respectively, and G_{inter} is the union of set of intraprocedural graphs of the functions in the program.

IECFG Construction. Alg. 12 (*BuildInterECFG*) shows the algorithm for constructing the interprocedural graph from the intraprocedural graphs. Initially, the interprocedural graph consists of the union of all the intraprocedural graphs, constructed independently, as described in Sect. 4. In the next step, the call graph is consulted to determine the call targets for each call site. At each call site, three edges are added: (a) a call edge from call node to start node of the target’s intraprocedural graph, (b) a call-return edge from the normal exit

Algorithm 10: *VisitHandler***Input:** h : A handler where $h = (t\ v)\ b_1$ **Output:** $N_b \times N_e \times N_{unres}$

```

1 let  $(N_{b_1}, N_{e_1}, N_{unres_1}) = \text{VisitBlock}(b_1)$ ;
2  $N_{catch} = \{n_h\}$ ;  $E_{reg} = E_{reg} \cup \{(n_h, n_s) \mid n_s \in N_{b_1}\}$ ;
3  $N_b = \{n_h\}$ ;  $N_e = N_{e_1}$ ;  $N_{unres} = N_{unres_1}$ ;

```

Algorithm 11: *VisitFunction***Input:** f : Function where $f = t\ id\ (\overline{id})$ **Output:** $G_{intra_f} = \langle n_s, n_e, n_{except}, N, E \rangle$

```

1 let  $(N_{b_1}, N_{e_1}, N_{unres_1}) = \text{VisitBlock}(b_1)$ ;
2  $E_{reg} = E_{reg} \cup \{(n_s, n) \mid n \in N_{b_1}\} \cup \{(n, n_e) \mid n \in N_{e_1}\}$ ;
3 foreach  $n \in S_{exit}$  where  $S_{exit} = \{n \mid n \in N_{unres_1} \wedge ir(n) = \text{return}\}$  do
4    $E_{reg} = E_{reg} \cup \{(n, n_e)\}$ ;
5 end
6 foreach  $n \in S_{throw}$  where  $S_{throw} = \{n \mid n \in N_{unres_1} \wedge ir(n) = \text{throw}\}$  do
7    $E_{except} = E_{except} \cup \{(n, n_{except}_f, T_{prog}(ir(n)))\}$ ;
8 end
9 foreach  $n \in S_{goto}$  where  $S_{goto} = \{n \mid n \in N_{unres_1} \wedge ir(n) = \text{goto}\}$  do
10   $E_{reg} = E_{reg} \cup \{(n, n_{tgt}) \mid n_{tgt} \in \text{Node}(\text{Label}(ir(n)))\}$ ;
11 end
12 for  $n \in N_{ecret}$  do
13    $\tau_{ecret} = \text{ECR}_\Gamma(n_{ecret})$ ;
14    $E_{except} = E_{except} \cup \{(n, n_{except}_f, \tau_{ecret})\}$ ;
15 end
16 return  $\langle n_{b_f}, n_{e_f}, n_{except}_f, N, E \rangle$ 

```

Fig. 9. *VisitHandler* and *VisitFunction* routines for ECFG construction

node of the target’s intraprocedural graph to the call-return node of the function call, and (c) an exception edge from the exceptional-exit node of the target’s intraprocedural graph to the exceptional-call-return node in the graph. The exception edge is annotated with the union (\bigcup_Γ) of the exception information on incoming exception edges of the exceptional-exit node, which serves as the initial dataflow fact for the interprocedural exception analysis. Finally, the summary edges connecting the call node with the call-return and exceptional-call-return nodes are removed.

Interprocedural Exception Analysis. Given that the IECFG construction algorithm initially gives a safe overapproximation of the interprocedural exception flow, the goal of the interprocedural analysis is to refine the dataflow facts on the exception edges as precisely as possible. Alg. 13 shows the interprocedural exception analysis algorithm. A single top-down propagation pass on the call graph will not model exceptions precisely in the presence of recursive functions. Therefore, we need to perform a dataflow analysis. Our analysis operates only on the exceptional-exit and exceptional-call-return nodes, and their incoming and outgoing edges. The abstract domain is the Signed-TypeSet domain as defined in Sect. 3. The analysis is implemented using a worklist W_{list} , which initially has the set of exceptional-exit and exceptional-call-return nodes in reverse topological order on the CG_{SCC} , the directed acyclic graph of strong connected components formed from the call graph. Each iteration of the algorithm removes a node from W_{list} , applies a transfer function, updates its outgoing exception

Algorithm 12. *BuildInterECFG*

```

Input:  $p$ : Program
Output:  $IG = \langle N_s, N_e, N_{excepte}, G_{inter} \rangle$ 
1  $G_{inter} = \cup_{f \in p} G_{intra_f}$ ;
2 foreach calltriple  $(n_{call}, n_{cret}, n_{excepte})$  do
3    $F = CallTargets(ir(n_{call}))$ ;
4   for  $f$  in  $F$  do
5     let  $G_{intra_f} = \langle n_{s_f}, n_{e_f}, n_{excepte_f}, N_f, E_f \rangle$ ;
6     let  $E_c = E_c \cup \{(n_{call}, n_{s_f}), (n_{e_f}, n_{cret})\}$ ;
7     let  $\tau_{exit} = \bigcup_{e_p \in prede(n_{excepte_f})} exceptE_{\Gamma}(e_p)$ ;
8      $E_{excepte} = E_{excepte} \cup \{(n_{excepte_f}, n_{exceptepret}, \tau_{exit})\}$ ;
9   end
10   $E_c = E_c - \{(n_{call}, n_{cret})\}$ ;
11   $E_{excepte} = E_{excepte} - \{(n_{call}, n_{exceptepret})\}$ ;
12 end

```

edges with the new dataflow facts and adds the successors nodes to W_{list} , if the data flow information has changed. The algorithm is continued until the W_{list} is empty, at which point the algorithm terminates with a fixed point. Termination of the algorithm is guaranteed due to the fact that the set of exceptions is finite.

A map $exceptN_{\Gamma}$ defines the most recent dataflow information, an element from the Signed-TypeSet domain, corresponding to each exceptional-exit or exceptional-call-return node. Another map $exceptE_{\Gamma}$ is used to hold the exception annotation on each exception edge. It is initialized to the union (\bigcup_{Γ}) of the exception information on the incoming edges for each node, and is updated every time the result of \bigcup_{Γ} changes. The transfer functions for the exceptional-exit and exceptional-call-return nodes differ in how they update the exception annotation on the outgoing edges, once \bigcup_{Γ} is computed:

- For an *exceptional-exit node*, each outgoing edge’s exception annotation is replaced by the newly computed information at the exit node. This operation reflects the refined set of all of possible (uncaught) exception types that could be thrown from a function, represented in the Signed-TypeSet domain (Lines 8-12 in Alg. 13).
- For an *exceptional-call-return node*, each outgoing edge’s old exception annotation, is replaced by an intersection (\bigcap_{Γ}) of the old exception annotation with the new exception information available at the node. The intersection operation serves to narrow the set of exception types that was previously assumed for a function call, and hence, iteratively increases the precision of the interprocedural exception flow graph (Lines 13-19 in Alg. 13).

Uncaught Exceptions. At the end of the analysis, some of the exception edges in the IECFG will have empty exception annotation, which can be eliminated. Empty exception annotations are identified in two phases. The first phase can be done immediately after the analysis, in which those that use a positive sign ($(+, \{\})$) can be removed. The second phase removes an empty exception annotation that uses a negative sign, and requires conversion of the exception information from the Signed-TypeSet domain to the domain of positive set of types. Alg. 14 shows the conversion algorithm, which is done only once, after the

Algorithm 13. *InterProceduralExceptionAnalysis*

```

Input:  $I_G = \langle N_s, N_e, N_{excep}, G_{inter} \rangle$ 
Input:  $I_{G'} = \langle N_s, N_e, N_{excep}, G_{inter'} \rangle$ 
1  $W_{list} = N_{excep} \cup \bigcup_{f \in M} N_{secret f}$ ;
2 while  $W_{list}$  is not empty do
3    $n \leftarrow \text{removeNode}(W_{list})$ ;
4    $\tau_{n_{old}} = \text{excep}N_{\Gamma}(n)$ ;
5    $\tau_{n_{new}} = \bigcup_{e_p \in \text{prede}(n)} \text{excep}E_{\Gamma}(e_p)$ ;
6   if  $\tau_{n_{old}} \neq_{\Gamma} \tau_{n_{new}}$  then
7     switch  $\text{typeOf}(n)$  do
8       case ExcepExit
9         foreach  $e_s \in \text{succe}(n)$  do
10           $\text{excep}E_{\Gamma}(e_s) = \tau_{n_{new}}$ ;
11           $W_{list} = W_{list} \cup \{\text{dst}(e_s)\}$ ;
12        end
13       case ExcepCallReturn
14         foreach  $e_s \in \text{succe}(n)$  do
15           $\text{excep}E_{\Gamma}(e_s) = \tau_{n_{new}} \cap_{\Gamma} \text{excep}E_{\Gamma}(e_s)$ ;
16          if  $\text{typeOf}(\text{dst}(e_s)) = \text{ExcepExit}$  then
17             $W_{list} = W_{list} \cup \{\text{dst}(e_s)\}$ ;
18          end
19        end
20      end
21     $\text{excep}N_{\Gamma}(n) = \tau_{n_{new}}$ ;
22  end
23 end
24 end

```

analysis is performed and also serves as a checker for the “no throw” guarantee. The algorithm walks the CG_{SCC} in reverse topological order, and at each step, uses the set of all exception types (positive) that could potentially be thrown by the transitive callees of a function, to serve as the universal set, from which to subtract the negated set of exceptions corresponding to the current function. Whenever a SCC of mutually recursive functions is encountered, the union of the set of uncaught exception types of each constituent function in the SCC is used as a sound overapproximation for the subtrahend. The algorithm, produces a map excep_F that gives for each function the set of potentially uncaught exception types.

Example. Fig. 10 shows the final interprocedural exception control flow graph for our example. The exception edges: $\text{ecr-readLine} \rightarrow \text{exe-get}$ and $\text{ecr-read} \rightarrow \text{exe-readLine}$ are notably missing from the graph. The analysis is able to infer this after performing the intersection operation, between exception information on incoming and outgoing edges of ecr-readLine and ecr-read :

$$(+, \{IOException\}) \cap_{\Gamma} (-, \{IOException\}) = (+, \{\}) \text{ and}$$

$$(+, \{EOFException\}) \cap_{\Gamma} (-, \{EOFException\}) = (+, \{\}).$$

However, we see that the program may potentially fail with the uncaught exception `std::bad_alloc` thrown in `get()` by the *new* operator. ■

Algorithm 14. *ComputeUncaughtExceptions*

```

Input:  $I_G = \langle N_s, N_e, N_{exception}, G_{inter} \rangle$ 
Output:  $except_F : F \mapsto T_{prog}$ 
1 let  $T_{except} = \{\}$ ;
2 for  $n_t \in N_{throw}$  do
3    $except_T(n_t) = T_{prog}(ir(n_t))$ ;
4    $T_{except} = T_{except} \cup T_{prog}(ir(n_t))$ 
5 end
6 for external function  $f$  do
7    $except_F(f) = \{t_{unknown}\}$ 
8 end
9  $F_l = ReverseTopoOrder(CG_{SCC})$ ;
10 while  $F_l$  is not empty do
11    $F_{scc} = RemoveFront(F_l)$ ;
12   for  $f \in F_{scc}$  do
13      $except_F(f) = \bigcup except_T(n_t)$  where  $(n_t \in N_{throw}) \wedge (\exists g \cdot g \in F_{scc} \wedge ir_n(n_t) \in g)$ 
14   end
15   for  $f \in F_{scc}$  do
16     switch  $except_{N_T}(n_{exception})$  where  $n_{exception} = N_{exception}(f)$  do
17       case  $(+, T_E)$ 
18          $except_F(f) = T_E$ 
19       case  $(-, T_E)$ 
20         foreach  $g \in TransitiveCallees(f)$  do
21            $except_F(f) = except_F(f) \cup except_F(g)$ ;
22         end
23          $except_F(f) = except_F(f) - T_E$ ;
24       end
25   end
26 end
27 end

```

6 Generating an Exception-Free Program

In this section, we describe our lowering algorithm that translates a given C++ program into a semantically equivalent program without exception-related constructs such as throw, catch, etc. The lowering algorithm uses the IECFG to eliminate exception-related constructs. There are two main distinguishing features of our lowering algorithm compared to existing C++ compilers:

- Our approach uses a combination of stack storage and reference parameters to simulate exceptions without generating additional runtime calls whose semantics have to be taught to existing static analysis tools.
- Our approach uses the exception target information available in the IECFG, and therefore, when compared to existing C++ lowering techniques, generates fewer infeasible edges between throw statements and catch blocks that are not present in the original program. It also handles insertion of destructors correctly. The modular design of the IECFG makes it easy to insert the destructor calls in a single pass.

The main steps of the lowering algorithm are as follows:

1. **Creation of Local Exception-Objects and Formal Parameters:** Each function's (say f) local variable list is extended with: (1) a "type-id" variable, and (2) a local exception-object variable for every exception type that can potentially be thrown within f . The "type-id" variable holds the type of the

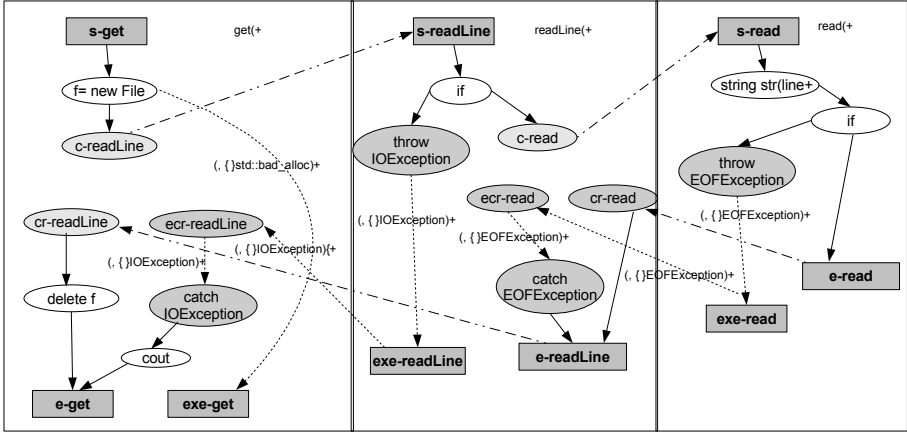


Fig. 10. Interprocedural Exception Control Flow Graph for the program in Fig. 11

thrown exception, and the local exception-object variable holds the thrown exception object and acts as storage for interprocedural exception handling. Additional parameters are added to f 's signature: (1) a reference parameter for every uncaught exception type that propagates out of the function, and (2) a reference parameter to hold the "type-id". These reference parameters propagate information about uncaught exceptions to a caller. At each call-site of f , appropriate local exception-objects and the caller's local type-id are passed additionally as parameters to f .

2. **Lowering throws and catch:** Based on the targets of throw statements in the IECFG, calls to the destructors of appropriate set of local objects are inserted. The thrown object is assigned to the local exception-object of the appropriate type and the local type-id variable is set to the thrown type. A goto is then inserted either to a catch block or to the exceptional-exit node. At the catch block, the local exception-object is assigned to the argument of the catch-header.
3. **Lowering exceptional-call-return nodes:** A *switch* statement (modeled using *ifs* in our IL) on the local type id is inserted, with one nested *case* for every uncaught exception type in the callee. The target node information from the IECFG is used to place calls to destructors of appropriate stack-allocated objects, for each *case*. Finally, a *goto* to the target (either a catch or an exceptional-exit) is inserted.
4. **Lowering exception exit node:** The local exception-objects and type-ids are copied into corresponding formal parameters. This serves to copy the exception objects out of the callee into the caller, which deals with the uncaught exceptions at its exceptional-call-return node.

The semantics preserving nature of the lowering algorithm can be established as follows. Our lowering mechanism is based on the observation that exception

handling preserves functional scoping even though exceptions result in non-local control flow. This is because the program has to unwind the call stack to invoke the destructors of local objects that have been constructed in the functions on the call stack until the exception is caught. Therefore, it largely mirrors the flow that happens during a regular call return. Our lowering mechanism mimics this flow by placing the destructor calls before the exceptional return of every function and passing pending exception objects through the additional reference parameters that were added by the lowering.

Subtleties introduced by some C++ features are handled as follows:

Throwing Destructors. As per the C++0x standards draft, destructors throwing an exception during stack unwinding result in a call to `std::terminate()`, which by default terminates the program. However, the destructor's callees can throw exceptions as long as they do not flow out of the destructor. Multiple live exceptions arising out of this are correctly handled in our lowering algorithm by the use of (a) local exception objects, which implicitly helps to maintain a stack of multiple outstanding exceptions, and (b) a global exception flag to detect a throwing destructor instance and trigger a call to `std::terminate()`.

Virtual Functions throwing different exceptions. Multiple function targets at a call site, quite common in C++ due to virtual functions, can in general throw different exception types. Our lowering algorithm prevents ambiguity in the function signature by generating a uniform interface at the call site, that uses the union of exception types that can be thrown by each possible target of a virtual function call.

Catch-all and rethrow. A catch-all clause (`catch (...)`) does not statically indicate the type of C++ exception handled by the clause. Rethrow statements (`throw;`) do not have a throw expression as an argument. Our lowering algorithm requires type and variable information, which is obtained by using the exception information from the IECFG. Since the IECFG has an edge to a catch clause annotated with the type of each possible exception thrown, the catch-all clause is expanded to a sequence of concrete clauses. Rethrows are handled by using the exception information from the nearest enclosing catch clause.

Exception Subtyping. The lowering algorithm assumes that the type of thrown exception is the same as the type of the catch clause, which may not be true in general due to the exception subtyping rules of C++. This case is handled by generating super class (w.r.t exception subtyping rules) local and formal exception objects, and assigning into them, thrown exception objects which are subclasses of the superclass object.

Example. The lowered code for our running example is shown in Fig. 11 after performing copy propagation to remove redundant local objects. Fig. 11 also shows the exception specifications for functions. (The specification has details of the pending call stacks for each uncaught exception, but is not shown here.) ■

<pre>string File::read(EOFException& e1, ExceptId& id) { string str(__line__); if (EOF) { EOFException tmp; e1 = tmp; id = EXCEP_EOF_EXCEPTION; ~str(); goto EExit_1; } return str; EExit_1: return string(); }</pre>	<pre>// EXCEPTION SPECIFICATIONS: // void get() throw (std::bad_cast); // string read() throw (EOFException); // string readLine() throw (IOException); #define EXCEP_NULL 0 #define EXCEP_BAD_CAST 1 #define EXCEP_IO_EXCEPTION 2 #define EXCEP_EOF_EXCEPTION 3 class EOFException { ... }; class IOException { ... };</pre>
<pre>void get (std::bad_cast& e3, ExceptId &id) { string s; IOException e4; ExceptId lid = EXCEP_NULL; File *file = new_alloc("1.txt"); if (file == NULL) { std::bad_cast tmp; e3 = tmp; id = EXCEP_BAD_CAST; goto EExit_2; } file->readLine(e4, lid); switch (lid) { case EXCEP_IO_EXCEPTION: goto Catch_L2; break; default: break; } delete file; return; Catch_L2: { cout << "IO-Failure"; } EExit_2: }</pre>	<pre>string File::readLine(IOException& e2, ExceptId& id) { string s; EOFException e1; ExceptId lid = EXCEP_NULL; if (invalidFile) { IOException tmp; e2 = tmp; id = EXCEP_IO_EXCEPTION; ~s(); goto Exit_2; } s = read(e1, lid); switch (lid) { case EXCEP_EOF_EXCEPTION: goto Catch_L1; break; default: break; } return s; Catch_L1: { EOFException& e = e1; return string(""); } Exit_2: return string(); }</pre>

Fig. 11. Exception specifications and exception-free program for the running example



Fig. 12. Exception Analysis and Transformation Workflow

7 Implementation and Experiments

We have implemented our exception analysis and transformation algorithms in an in-house extension of CIL [14], which handles C++ programs. The exception analysis implementation has about 6,700 lines of OCAML code. Fig. 12 shows the workflow for analyzing and transforming C++ programs with exceptions. The given C++ program is initially parsed by our frontend into a simplified intermediate version of C++ (IR0) similar to the IL shown in Section 2. The IR0 code is then fed to our interprocedural exception analysis and transformation framework, which produces lowered C++ code without exceptions. The

Table 1. Results of interprocedural exception analysis and exception safety checks

Benchmark	Simplified LOC	ECFG Build Time(s)	IECFG Build & Analysis Time(s)	#Excep Edges before Analysis	#Excep Edges after Analysis	“No throw” guarantee Coverage (% Functions)	“No leak” check results (#detected /#actual)
multiple-live	479	0.01	0.01	10	6	71 %	0/0
ctor-throw	585	0.02	0.02	33	11	89 %	1/1
recursive	643	0.02	0.03	27	17	64 %	0/0
shared-inherit	667	0.02	0.04	59	27	71 %	1/1
bintree-duplicate	770	0.06	0.05	31	13	91 %	0/1
list-baditerator	784	0.04	0.04	36	17	79 %	0/2
virtual-throw	809	0.02	0.03	39	28	46 %	0/4
nested-try-catch	809	0.03	0.03	33	17	68 %	2/2
loop-break-cont	814	0.04	0.03	33	17	68 %	2/2
nested-rethrow	820	0.04	0.03	35	19	68 %	4/4
new-badalloc	849	0.02	0.03	30	15	76 %	2/2
template	860	0.03	0.04	51	28	67 %	1/1
dyn-cast	872	0.03	0.05	35	16	81 %	1/1
iolib	919	0.01	0.01	9	7	40 %	1/1
delegat-dtor-throw	1305	0.04	0.05	63	62	53 %	0/0
std-uncaught-dtor	1348	0.05	0.07	73	71	58 %	1/1

transformed C++ code is then lowered to C by a module that lowers various object-oriented features into plain C. The C++-to-C lowering module transforms features such as inheritance and virtual-function calls without the use of runtime structures such as virtual-function and virtual-offset tables. Therefore, the lowered source code is still at a relatively high-level for further static analysis. The lowered C code is then fed into F-SOFT [7], where standard bug detection and verification tools that work on C are applied.

We have evaluated our exception analysis and transformation algorithms on a set of C++ programs [13]. The programs test usage of various C++ exception features in realistic scenarios, some of which are close to standard C++ collection class usage [21]. We used the results of our analysis to test the “no throw” guarantee, immediately before lowering, and the results of our transformation to test the “no leak” guarantee using F-SOFT. For the experiments, exceptions of type *tunknown* from external library calls were omitted. Tab. 1 shows the results. The running time for ECFG construction for all programs is low, while the IECFG construction and analysis is quite comparable, with most of the time spent in the interprocedural exception analysis. Our interprocedural exception analysis is able to achieve an average reduction of about 38% in the number of exception edges, with the IECFG constructed immediately before the interprocedural analysis serving as the baseline. On an average, around 66% of the functions in a program were certified as “no throw”.

The last column shows the results of running F-SOFT, specifically a memory leak detector module, on the lowered programs. 13 of the 16 benchmarks that we used for these experiments had memory leaks along exception paths, and F-SOFT reported all memory leaks in 10 of the 13 benchmarks. F-SOFT failed to find memory leaks for 3 benchmarks due to timeouts and reported bogus witnesses only for `new-badalloc` due to the limitation of our in-house C-lowering. For these experiments we used a time-bound of 10 minutes for the verification.

Table 2. Results of interprocedural exception analysis on open-source benchmarks

Open-source Benchmark	Simplified LOC	ECFG Build Time(s)	IECFG Build & Analysis Time(s)	#Excep Edges before Analysis	#Excep Edges after Analysis	“No throw” guarantee Coverage (% Functions)
tinyxml	4884	0.39	1.41	1204	830	74 %
mailutils	8365	0.19	0.36	494	316	78 %
coldet	8422	0.27	0.22	591	20	98 %
id3lib	14070	1.73	4.18	2091	372	93 %

One of the reasons for the timeouts is that the lowering algorithm generates programs that is atypical of the C source code that F-SOFT has previously analyzed, which affects the performance of the model checker. As an example, we have found that the addition of destructor calls during stack unwinding on exceptional edges introduces many additional destructor call sites; in the benchmark `std-uncaught-dtor`, there were 31 call sites to a particular class destructor. The additional destructor calls during stack unwinding yield function call graphs that are very different from what F-SOFT usually encounters. Therefore, additional heuristics, such as selective function inlining for destructor calls, will likely improve the performance of the model checker on the models generated by the exception analysis module.

Results on open-source benchmarks. We have also applied the IECFG construction algorithm on a set of open-source benchmarks shown in Tab. 2. The coldet benchmark is an open source collision detection library used in game programming. GNU mailutils is an open source collection of mail utilities, servers, and clients. TinyXML v2.5.3 is a light-weight XML parser, which is widely used in open-source and commercial products. The open source library id3lib v3.8.3 is used for reading, writing, and manipulating ID3v1 and ID3v2 tags, which are the metadata formats for MP3s. Tab. 2 shows the reduction in the number of exception edges due to our interprocedural analysis. A direct consequence of this reduction is seen in the “no throw” guarantee numbers, which represent the percentage of the total functions in the program, for which we are able to guarantee that no exceptions will be thrown by them. For coldet, which had the maximum reduction in the number of edges, the number of functions guaranteed not to throw is about 98%. We are encouraged by the results of our experiments on the open source benchmarks. For these benchmarks, the time taken to compute the IECFG is less than 5s. Therefore, we believe that the analysis will scale to even larger examples.

Memory leaks in mailutils applications. We also applied the memory-leak checker module of F-SOFT on two applications that use the mailutils library: (1) `iconv`, which converts strings from one character encoding to another using the mailutils library, and (2) `murun`, which tests the various kinds of streams in the mailutils library. F-SOFT reported one memory leak in `iconv` and three memory leaks in `murun` involving exceptional control flow. The offending code snippet in `iconv` is shown in Fig. 13. In the `try` block, the invocation of the constructor

`FilterIconvStream()` for variable `cvt` may throw an exception. However, when the exception is handled by the catch block, the memory allocated at the start of the try block is not deallocated, which results in a memory leak. The leaks reported in `murun` also have a similar flavor.

In addition, F-SOFT reported a leak in `iconv` where the memory allocated in the constructor of class `FilterIconvStream` is never deallocated. Note that this leak occurs even when no exceptions are thrown by the application.

```

...
try {
    StdioStream *in = new StdioStream (stdin, 0);
    in->open ();
    FilterIconvStream cvt (*in, (string)argv[1], (string)argv[2], 0,
        mu_fallback_none);
    cvt.open ();
    delete in;
    ...
}
catch (Exception& e) {
    cerr << e.method () << ":_" << e.what () << endl;
    exit (1);
}

```

Fig. 13. Memory leak along an exception path in `iconv`

8 Related Work

Most related work deals with exceptions in Java. Earlier, we discussed many differences between exceptions in C++ and Java, thus requiring different approaches. Sinha and Harrold [16] incorporate the control flow effects due to explicit Java exceptions in an interprocedural control flow graph (ICFG) using a flow-sensitive type analysis, and discuss their applications to control dependence analysis and slicing. The ICFG used in their analysis has no exceptional-call-return node and can have multiple exceptional-exit nodes in a function. In contrast, our IECFG is modular and has an exceptional-call-return node for every function call, which is required for modeling implicit C++ destructor calls. Jo et al. [8,9,3] construct an exception flow graph for Java, using a set constraint analysis that is required to iterate to convergence. Gherghina and David [6] present a specification logic for exceptions for Java-like languages and verify exception-safety guarantees. Their specification logic does not model destructors along exception paths because they target Java-like languages, and therefore, cannot be used for verifying C++ programs. Mao and Lu [12] perform the analysis for C++, without explicitly modeling destructors. Robillard and Murphy [15] develop an analysis that handles both checked and unchecked exceptions in Java. In contrast to all these approaches, our analysis based on the Signed-TypeSet domain can be terminated safely at any point after the IECFGs have been constructed, thus permitting sound static analysis on subparts of the IECFG. Given the prevalence of separate compilation in large systems, a modular approach that

can be safely terminated at any step is essential for scalability and adoption into compilers. The Signed-TypeSet domain utilizes a similar idea as was used in the form of *difference sets* for class hierarchy analysis [4].

Weimer and Necula [20] propose an intraprocedural, path-sensitive analysis for checking typestate specifications along exception paths in Java. Buse and Weimer [2] propose a similar symbolic analysis for automatic documentation of Java exceptions. Bravenboer and Smaragdakis [1] propose a solution for Java, where pointer analysis and exception analysis problems are framed and solved in a mutually recursive manner, with each improving the precision of the other. Fu and Ryder [5] propose a static analysis that computes chains of semantically related exception flow links, by composing existing exception analyses to give longer exception paths. Our IECFG construction and exception analysis for C++, could potentially be enhanced with all of the above techniques to improve the accuracy of our exception model. Li et al. [11], propose a technique to check the “no leak” guarantee for Java, using a combination of static analysis and model checking. Similarly, Torlak and Chandra present an interprocedural static analysis algorithm to detect resource leaks in Java programs [19]. In our work, we use F-SOFT [7] for checking for resource leaks, following our exception analysis and lowering transformation.

9 Conclusions and Future Work

This paper introduced an interprocedural analysis framework for accurately modeling C++ exceptions. In this framework, control flow induced by exceptions is captured in a modular interprocedural exception control flow graph (IECFG). This graph is refined by a novel dataflow analysis algorithm, which abstracts the types of exception objects over a domain of signed set of types. Unlike exception analyses proposed elsewhere for other languages, this analysis can be safely terminated at well-defined points during interprocedural propagation, thereby allowing clients to trade-off speed over precision. The paper then presented a lowering transformation that uses the computed IECFG to generate an exception free program. This transformation is designed specifically to permit easier and more precise static analysis on the generated code. Finally, the paper demonstrated two applications of the framework: (a) automatic inference of exception specifications for C++ functions and (b) checking the “no throw” and “no leak” exception safety properties.

In the future, we intend to perform additional experiments on larger benchmarks. As shown in Sect. 7, the IECFG computation presented here should scale well for larger benchmarks. Finally, we are investigating to selectively allow conditional exception edges during the IECFG construction. Such conditional exception edges could be used to model cases involving throwing destructors or other standard library objects such as `cout` more precisely. For example, a throwing destructor would be allowed to propagate the fact that `std::uncaught_exception()` was queried before throwing an exception. This could be used to eliminate spurious calls to `std::terminate()` when returning from such destructors. Similarly, we can annotate other calls, such as uses of `cout` with the information that it may

throw an exception, if the surrounding context had set the relevant information using the `ios::exceptions()` method. Such selective guards on exception edges would not substantially decrease the performance of the analysis but would allow further reduction of computed exception-catch links.

References

1. Bravenboer, M., Smaragdakis, Y.: Exception analysis and points-to analysis: Better together. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 1–12. ACM, New York (2009)
2. Buse, R.P., Weimer, W.R.: Automatic documentation inference for exceptions. In: ISSTA, pp. 273–282. ACM, New York (2008)
3. Chang, B.-M., Jo, J.-W., Yi, K., Choe, K.-M.: Interprocedural exception analysis for Java. In: Proc. of Symp. on Applied Computing, pp. 620–625 (2001)
4. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995)
5. Fu, C., Ryder, B.: Exception-chain analysis: Revealing exception handling architecture in Java server applications. In: ICSE, pp. 230–239 (May 2007)
6. Gherghina, C., David, C.: A specification logic for exceptions and beyond. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 173–187. Springer, Heidelberg (2010)
7. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M., Kahlon, V., Wang, C., Yang, Z.: Model checking C programs using F-Soft. In: IEEE International Conference on Computer Design, pp. 297–308 (October 2005)
8. Jo, J.-W., Chang, B.-M.: Constructing Control Flow Graph for Java by Decoupling Exception Flow from Normal Flow. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3043, pp. 106–113. Springer, Heidelberg (2004)
9. Jo, J.-W., Chang, B.-M., Yi, K., Choe, K.-M.: An uncaught exception analysis for Java. *Journal of Systems and Software* 72(1), 59–69 (2004)
10. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International Symposium on Code Generation and Optimization (CGO), Palo Alto, California (March 2004)
11. Li, X., Hoover, H., Rudnicki, P.: Towards automatic exception safety verification. In: Proc. of Formal Methods, pp. 396–411. Springer, Heidelberg (2006)
12. Mao, C.-Y., Lu, Y.-S.: Improving the robustness and reliability of object-oriented programs through exception analysis and testing. In: IEEE International Conference on Engineering of Complex Computer Systems, vol. 0, pp. 432–439 (2005)
13. NECLA verification benchmarks,
http://www.nec-labs.com/research/system/systems_SAV-website/benchmarks.php
14. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Int. Conf. on Comp. Construct, pp. 213–228. Springer, Heidelberg (2002)
15. Robillard, M.P., Murphy, G.C.: Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering Methodologies* 12(2), 191–221 (2003)
16. Sinha, S., Harrold, M.J.: Analysis and testing of programs with exception handling constructs. *IEEE Trans. on Software Engineering* 26, 849–871 (2000)

17. C.standards committee. Working draft, standard for programming language C++ (2010), <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3126.pdf> (accessed September 26, 2010)
18. Stroustrup, B.: Exception safety: Concepts and techniques. In: Romanovsky, A., Cheraghchi, H.S., Lee, S.H., Babu, C. S. (eds.) ECOOP-WS 2000. LNCS, vol. 2022, pp. 60–76. Springer, Heidelberg (2001)
19. Torlak, E., Chandra, S.: Effective interprocedural resource leak detection. In: Int. Conf. on Softw. Eng., pp. 535–544. ACM, New York (2010)
20. Weimer, W., Necula, G.C.: Exceptional situations and program reliability. ACM Trans. Programming Languages and Systems 30(2), 1–51 (2008)
21. Weiss, M.A.: Data Structures and Algorithm Analysis in C++. Addison-Wesley Longman Publishing Co., Inc., Boston (1998)

Detecting and Escaping Infinite Loops with Jolt

Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard

Massachusetts Institute of Technology, Cambridge, MA, USA
{mcarbin,misailo,mkling,rinard}@csail.mit.edu

Abstract. Infinite loops can make applications unresponsive. Potential problems include lost work or output, denied access to application functionality, and a lack of responses to urgent events. We present *Jolt*, a novel system for dynamically detecting and escaping infinite loops. At the user's request, Jolt attaches to an application to monitor its progress. Specifically, Jolt records the program state at the start of each loop iteration. If two consecutive loop iterations produce the same state, Jolt reports to the user that the application is in an infinite loop. At the user's option, Jolt can then transfer control to a statement following the loop, thereby allowing the application to escape the infinite loop and ideally continue its productive execution. The immediate goal is to enable the application to execute long enough to save any pending work, finish any in-progress computations, or respond to any urgent events.

We evaluated Jolt by applying it to detect and escape eight infinite loops in five benchmark applications. Jolt was able to detect seven of the eight infinite loops (the eighth changes the state on every iteration). We also evaluated the effect of escaping an infinite loop as an alternative to terminating the application. In all of our benchmark applications, escaping an infinite loop produced a more useful output than terminating the application. Finally, we evaluated how well escaping from an infinite loop approximated the correction that the developers later made to the application. For two out of our eight loops, escaping the infinite loop produced the same output as the corrected version of the application.

1 Introduction

From: "Armando Solar-Lezama" <asolar@csail.mit.edu>
To: "Martin Rinard" <rinard@csail.mit.edu>
Subject: Thanks

I was writing a document in Word this morning, and after about an hour of unsaved work, Word went into an infinite loop that made the application completely frozen. So, having listened to your talks too many times, I got my debugger, paused the program, changed the program counter to a point a few instructions past the end of the loop, and let it keep running from there. Word went back to working as if nothing had ever happened. I was able to finish my document, save it, and close Word without problems.

So thanks,
Armando.

As the above email illustrates, infinite loops can make an application unresponsive to its users. The potential consequences include loss of work or an inability to use the application for its intended purpose.

One potential solution (as deployed by Professor Solar-Lezama above) is to drop the application into a debugger, find the infinite loop, then move the program counter past the end of the loop, thereby enabling the application to continue its productive execution. Unfortunately, not everyone has the technical skill to use this solution. And even if one does, using the debugger, finding the loop, and moving the program counter past the end of the loop can be a tedious and annoying process.

1.1 Automatic Detecting and Escaping Infinite Loops

We present *Jolt*, a novel system for detecting and (if desired) escaping infinite loops. If a user suspects that an application may be in an infinite loop, he or she can instruct Jolt to monitor the execution of the application. Specifically, Jolt records the program state at the start of each loop iteration. The next time execution reaches the start of the loop, Jolt compares the current state to the saved state. If the current and saved states are the same, then the loop has made no progress and Jolt has detected an infinite loop. At the user's option, Jolt can escape the loop (i.e., transfers control to a statement after the loop to enable the application to continue its execution beyond the loop). The immediate goal of the continued execution is to enable the application to save any pending work, finish any pending computation, or respond to any urgent events. Ideally, escaping the loop would also enable the application to continue its normal execution indefinitely.

1.2 Evaluation

We evaluated Jolt by applying it to eight infinite loops in five applications (ctags, grep, indent, look, ping). We attached Jolt to each of these applications while they were executing on inputs that triggered the infinite loops. Jolt successfully detected seven of the eight infinite loops; the remaining loop changes the state on every iteration (Jolt is designed to detect only infinite loops in which the program state does not change across iterations).

As part of each case study, we used Jolt to exit the infinite loops and observed the resulting continued execution. In general, the applications are structured to process multiple input units (such as lines, modules, or records). The infinite loops occur when one of the input units hits a corner case in the application's code. Escaping the loop typically causes some perturbations in the computation on the current unit. But, by the time the application starts processing the next unit, it has recovered and is able to process this unit with no problems (unless, of course, this unit also triggers the infinite loop). The end result is that the application is often able to produce largely or even fully useful output.

We note that a similar phenomenon is partially responsible for the effectiveness of failure-oblivious computing [28] and SRS crash suppression [22] in enabling applications to recover from memory errors — because the applications

tend to have short error propagation distances, errors that occur when processing one unit tend not to affect the processing of the next unit.

We also compared the output of escaping infinite loops with that of simply terminating the application at the infinite loop (for example, by hitting Ctrl-C). Terminating the application, of course, leaves it unable to process subsequent input units. And in some cases, the application produces no output at all — it is designed to produce all of its output after it has processed all of the input units. We found that for all of our applications, escaping the infinite loop produced a more useful output than terminating the application.

Finally, we acquired versions of the applications that were corrected by their developers. We then compared the outputs of escaping our infinite loops with the outputs of these versions of the applications. In two out of our eight infinite loops, escaping a loop produced an output that is identical to the output of the fixed version of the application. For the remaining infinite loops, output degradation was limited to the portion of the output that was generated from the input unit that caused the infinite loop.

1.3 Contributions

This paper makes the following contributions:

- **Detection and Escape:** It presents a system, Jolt, for detecting and (if desired) escaping infinite loops. Our technique uses both static source code instrumentation and dynamic binary instrumentation. Jolt statically instruments the source of an application with runtime calls that demarcate the entry, exit, and body of every loop in the control flow graph of each function in the program.

When instructed by a user, Jolt dynamically attaches to a running instance of the application and inserts instrumentation to record the state at the start of each loop iteration. As the application executes, Jolt compares the current state with the state from the previous iteration. If the states are equal, Jolt has detected an infinite loop. At the user's option, Jolt can then escape and continue execution at a statement following the loop.
- **Detection Evaluation:** It presents empirical results from applying Jolt to eight loops in five applications. Jolt detects seven of the eight loops (the remaining loop changes the state on every iteration). It also presents an evaluation of the performance of our technique; it imposes no more than 8.6% overhead on our applications when Jolt is not monitoring the application. And, when monitoring, Jolt detected all infinite loops in less than 1 second.
- **Escape Evaluation:** It presents empirical results that demonstrate that for all of our benchmark applications, escaping an infinite loop produces a more useful output than terminating the application. Moreover, escaping an infinite loop produces an output that is identical to the output of a manually fixed version of the application for two out of our eight infinite loops. In general, continued execution after the loop is successful because the applications tend to have short error propagation distances.

In our opinion, our results support the hypothesis that Jolt can provide a useful alternative to simply terminating the application when it encounters an infinite loop. We anticipate that Jolt will prove to be useful for interactive applications in which terminating the application would cause the user to lose work or leave the user without useful output. More generally, we expect that Jolt may also enable a wide range of applications to provide useful service even in the presence of infinite loops that would, in the absence of Jolt, render the application completely unresponsive.

2 The Jolt System

To provide users with a low-overhead system for infinite loop detection and escape, we have designed Jolt around two components:

Compiler: Jolt’s compiler enables a developer or user to compile the source code of his or her application to obtain a binary executable that is amenable to infinite loop detection. In particular, Jolt’s compiler adds lightweight instrumentation to the source of the application to identify the boundaries of loops, which can be difficult to identify accurately from a binary executable [15,34].

Detector: Jolt’s detector can, at the user’s request, dynamically attach to and analyze a running instance of an application that the user believes is caught in an infinite loop (if the application has been compiled with Jolt’s compiler). If the detector determines that the application is caught in an infinite loop, it presents the user with the option to escape the loop.

2.1 Example

To illustrate how Jolt compiles and analyzes an application, we present an in-depth example of applying Jolt to an infinite loop in `ctags`, one of our benchmark applications.

`Ctags` scans program source files to produce an index that maps program entities (e.g., modules, functions, and variables) to their line numbers within the source files [1]. `Ctags` contains multiple modules for parsing and extracting the index, each of which is specific to a particular programming language. An integrated development environment can later use such an index file to allow programmers to quickly navigate to the definitions of modules, functions, and other program entities by name.

Figure 1 presents a Python code snippet taken from the `numpy` numerical matrix manipulation routine library. `Ctags` was designed to parse this source code and output an index, which indicates that, e.g., the function `get_pkgdocs()` begins on Line 1.

This code snippet uses multi-line strings (which are delimited by matched pairs of triple-quote literals, `'''` or `"""`, and can span more than one line) on Lines 3 and 4 to construct the string `retstr`. The backslash between the two

```

1 def get_pkgdocs(self):
2     if symbols:
3         retstr += """\n\nGlobal symbols from subpackages"" \
4             """\n-----\n"" + \
5         self._format_titles(symbols, '-->')

```

Fig. 1. Example Python Code

lines is admissible Python syntax and appears in the original file; in Python two lines that are separated by a backslash are treated as a single line. As a consequence, ctags merges the two lines into a single line during its preprocessing stage. However, when multiple multi-line strings appear on the same line in a Python source file, ctags version 5.7beta can enter an infinite loop.

2.2 Infinite Loop

Figure 2 presents `find_triple_end()`, the function from Ctags's Python module that loops infinitely on the code snippet from Figure 1. The function determines if `string`, which points to a character buffer containing a single line of text from a parsed file, closes an already open multi-line string. The parameter `which` contains the delimiter that began the multi-line string (either `'''` or `"""`).

At the beginning of each iteration of the loop, `s` points to some position in `string` and `which` contains the triple-quote that began the last multi-line string. Within the loop, if `s` does not contain a matching triple-quote, then the loop exits (Line 5). If `s` does contain a matching triple-quote, then the computation 1) records that the currently opened multi-line string has been closed, by setting `which` to `NULL` on Line 7, and 2) checks if `s` contains any additional triple-quotes.

If `s` does not contain an additional triple-quote, then the computation exits the loop (Line 9). Otherwise, the computation 1) records that a new multi-line string has been opened (by updating `which` in `find_triple_string()`), and

```

1 static void find_triple_end(char const *string, char const **which) {
2     char const *s = string;
3     while (1) {
4         s = strstr (string, *which);
5         if (!s) break;
6         s += 3;
7         *which = NULL;
8         s = find_triple_start(s, which);
9         if (!s) break;
10        s += 3;
11    }
12 }

```

Fig. 2. Source Code for Ctags

```

1 #define LOOP_ID 148
2
3 static void find_triple_end(char const *string, char const **which) {
4     char const *s = string;
5
6     jolt_loop_entry(LOOP_ID);
7     while (1) {
8         if (!jolt_loop_body(LOOP_ID)) {
9             goto jolt_escape;
10        }
11        s = strstr (string, *which);
12        if (!s) {
13            jolt_loop_exit(LOOP_ID);
14            break;
15        }
16        s += 3;
17        *which = NULL;
18        s = find_triple_start(s, which);
19        if (!s) {
20            jolt_loop_exit(LOOP_ID);
21            break;
22        }
23        s += 3;
24    }
25 jolt_escape:
26 }

```

Fig. 3. Instrumented Source Code for Ctags

2) updates `s` to point to the character after the newly found triple-quote. The computation then returns to the beginning of the loop to look for a triple-quote that closes the newly opened multi-line string.

The programmer wrote this loop with the intention that each iteration of the loop would start at some position in `string` (given by `s`) and either exit, or continue with another iteration that starts at a later position in `string`. To establish this, the value of `s` is incremented by the functions `strstr()` (Figure 2, Line 4) and `find_triple_string()` (Figure 2, Line 8).

However, in the call to `strstr()` the developer mistakenly passed `string`, instead of `s`, as the starting position for each iteration. As a consequence, every iteration of the loop starts over at the beginning of `string`, which can cause an infinite loop. For example, if the triple-quotes of the first and the second multi-line string are of the same type (as in Figure 1), then at the beginning of every loop iteration (except the first), the values of `s` and `which` are always the same: `s` equals to the starting position of the second multi-line string and `which` contains the triple-quote that starts the second multi-line string.

2.3 Compilation

Figure 3 presents the instrumentation that Jolt’s compiler adds to the source code of this loop in `ctags`. For every loop in the application, Jolt identifies and marks the following:

- **Loop ID:** Jolt gives each loop in the application a unique identifier (Line 1).
- **Loop Entry:** At the entry point of the loop, Jolt adds a call to the function `jolt_loop_entry()` to notify Jolt’s runtime that the application has reached the beginning of a loop (Line 6).
- **Loop Exit:** At each exit point from the loop, Jolt adds a call to the function `jolt_loop_exit()` immediately before exiting the loop to notify Jolt’s runtime that the application is about to exit a loop (Lines 13 and 20).
- **Loop Body and Loop Escape Edge:** Jolt adds a call to the function `jolt_loop_body()` at the start of the loop body to let Jolt control the execution of the loop (Line 8). If `jolt_loop_body()` returns true, then the application will execute the body of the loop. If `jolt_loop_body()` returns false, then the application will escape the loop by branching to the block immediately after the loop, which is marked by the label `jolt_escape` (Line 25). By default, `jolt_loop_body()` returns true if a user has not used Jolt’s detector to attach to the application.

After instrumenting `ctags` source code, Jolt uses the LLVM 2.8 compiler infrastructure [16] to compile the source code down to an executable (a 32-bit or 64-bit ELF executable in our current implementation). Though the instrumented executable incurs some overhead (Section 6), its semantics are exactly the same as that of the uninstrumented application — that is, until a user instructs Jolt’s detector to attach to a running instance of the application.

2.4 Detection

Once the user believes that `ctags` may be caught in an infinite loop, he or she can use Jolt’s user interface to scan the list of active system processes and select the suspect `ctags` process. When the user selects the process, Jolt’s infinite loop detector attaches to the running process and begins monitoring its execution.

Conceptually, Jolt records a snapshot of the state of the application at the beginning of each loop iteration. If `ctags` is caught in the infinite loop from Section 2.2, Jolt’s detector will recognize that 1) the application modifies only the variables `s` and `which`, and 2) that these variables have the same values at the beginning of each loop iteration. Given this observation, Jolt will report to the user that the application has entered an infinite loop.

2.5 User Interaction

After Jolt detects an infinite loop, it presents the user with the option to escape the loop. If the user chooses to escape the loop, he or she can place Jolt into one of two interaction modes:

- **Interactive Mode:** After Jolt forces the application to escape the loop, Jolt detaches from the application. Jolt will not detect any subsequent infinite loops unless the user again instructs Jolt to attach to the running application.
- **Vigilant Mode:** After Jolt forces the application to escape the loop, Jolt stays attached to the application. Jolt will continue to detect and escape infinite loops without further user interaction. Vigilant mode is useful when the application encounters an input that repeatedly elicits infinite loops.

It is also possible to support additional modes in which Jolt stays attached, but asks the user each time it detects an infinite loop before escaping the loop. Or, if Jolt is unable to detect an infinite loop, a user may, at his or her own discretion, choose to escape a loop that has been executing for a long time.

2.6 Escaping the Infinite Loop

Terminating ctags during the infinite loop from Section 2.2 would cause the user to lose some or all of the indexing information for the current file. Moreover, terminating ctags would leave it unable to process any subsequent files that could have been passed on the command line. If, instead, a user elects to escape the loop, then Jolt will force the application to exit the loop by returning `false` for the next call to `jolt_loop_body()`. As a consequence, ctags will terminate and produce a well-formed output. This output will include some of the definitions from the current file and all of the definitions from any subsequent files.

The quality of the output from the current file depends on the position of the triple-quote that closes the second string. If the triple-quote is on the same line (such as in Figure 1), then the quotes become unmatched, effectively causing ctags to treat the remainder of the current file as a multi-line string. On the other hand, if the triple-quote is on a subsequent line, then ctags will produce the exact same set of definitions as intended by the developers (which we verified by inspecting a later, fixed version of the application).

3 Implementation

Our design adopts a hybrid instrumentation approach that uses both static source code instrumentation and dynamic binary instrumentation. Jolt statically inserts lightweight instrumentation into the application to monitor the application’s control flow. Then, after it has attached to the application, Jolt inserts heavyweight dynamic binary instrumentation to monitor changes in the application’s state. Between these two components, our design balances Jolt’s need for precise information about the structure of an application with our desire to minimize overhead on the application when it is not being monitored.

3.1 Static Instrumentor

Jolt’s static instrumentor provides Jolt’s detector with control flow information that may otherwise be difficult to extract accurately from the compiled binary of

a program. The static instrumentor inserts function calls that notify the detector of the entry, body, and exit of each natural loop [21] in the control flow graph of each function in the program (as we presented in Section 2). Jolt currently does not instrument potential infinite recursions or unstructured loops that may occur because of exception handling or `gotos` that produce unnatural loops.

Jolt's static instrumentor also selects an escape destination for each loop in the application. The static instrumentor chooses an escape destination from one of the normal exit destinations of the loop. In general, a loop may contain multiple exit destinations (this can occur, for example, if the loop body uses `goto` statements to exit the loop). Jolt currently chooses the first loop exit as identified by LLVM.

It also possible for the loop to contain no exits at all. This can happen, for example, if the program uses an exception mechanism such as `setjmp/longjmp` to exit the loop. In this case Jolt inserts a return block that causes the application to return out of the current procedure. When Jolt escapes the infinite loop, it transfers control to this return block. Researchers have demonstrated that simply returning from a function can be an effective way to work around an error within its computation [30].

We have implemented the static instrumentor as an LLVM compiler pass that operates on LLVM bitcode, a language-independent intermediate representation. Given the instrumented bitcode of an application, we then use LLVM's native compiler to generate a binary executable.

3.2 Dynamic Instrumentor

When a user enables Jolt's infinite loop detection on a running program, Jolt's dynamic binary instrumentation component dynamically attaches the running program and inserts instrumentation code to record the state of the program as it executes. Jolt's instrumentation (conceptually) records, at the top of each loop, a snapshot of the state that the last loop iteration produced. To avoid having to record the entire live state of the application, Jolt instruments the application to produce a *write trace*, which captures the set of registers and addresses that each loop iteration writes.

Write Trace: Jolt instruments each instruction in the application that modifies the contents of a register or memory address. For each register or memory address that an instruction modifies, the instrumentation code dynamically records either the identifier of the register or the memory address into the write trace.

Snapshot: At the top of each loop, the inserted Jolt instrumentation uses the resulting write trace to record a snapshot. This snapshot contains 1) the list of registers and memory addresses written by the last iteration and 2) the values in those registers and memory addresses at the end of the last iteration. Jolt records a snapshot only if it has a complete write trace from the last loop iteration (the trace may be incomplete if the user attached Jolt to the application sometime during the iteration).

Library Routine Abstraction: To record a full write trace of an application, Jolt must instrument all of the application’s running code, including libraries. However, some libraries may modify internal state that is unobservable to the application. For example, some libc routines modify internal counters, (i.e., the number of bytes written to a file, or the number of memory blocks allocated by the program), that change after every invocation of the routine. If an application invokes one of these routines during a loop that is, otherwise, producing the same state on each iteration, then Jolt will be unable to detect the infinite loop. However, these counters are often either 1) not exposed to the application, or 2) exposed but not used by the application in a given loop. Therefore, we allow Jolt to accept a set of *library routine abstractions* to explicitly specify the set of observable side-effects of library routines.

A library routine abstraction specifies if the routine modifies observable properties of its arguments. For example, consider the `write` routine from libc:

```
ssize_t write(int filedes, const void *buf, size_t nbyte);
```

This function does not modify the contents of `buf`, but it does modify the current position of the file cursor, which the application can query by calling `ftell(filedes)`. If during an infinite loop, Jolt does not observe any calls from the application to `ftell(filedes)`, then Jolt can exclude the side-effects of a call to `write(filedes, ...)` from the snapshot.

We have implemented library routine abstractions for the subset of libc library calls that are invoked by our benchmark programs (e.g., `read`, `write`, `printf`). We anticipate that library routine abstractions need only be implemented for libraries that are considered a part of the runtime system of the application (e.g., allocation, garbage collection, and input/output routines).

Detection: At the beginning of each loop iteration, Jolt’s detector compares the snapshots of the two previously executed loop iterations. If the two snapshots are the same — i.e., both snapshots contain the same registers and memory addresses in their write traces and the recorded values for these registers and memory addresses in the snapshots are the same — then Jolt reports that it has detected an infinite loop.

We have implemented the dynamic instrumentor on top of the Pin dynamic program analysis and instrumentation framework [18]. Our use of Pin enables the dynamic instrumentor to analyze both Linux and Windows binaries that have been compiled for the x86, x64, or IA-64 architectures.

4 Empirical Evaluation

We next present a set of case studies that we performed to investigate the consequences of using Jolt to detect and escape infinite loops.

Table 1. Studied Infinite Loops

Benchmark	Version	Reference Version	Bug Report	Location
ctags-fortran	5.5	5.5.1	Ctags-734933	fortran.c, parseProgramUnit, 1931
ctags-python	5.7b (646)	5.7b (668)	Ctags-1988027	python.c, find_triple_end, 364
grep-color	2.5	2.5.3	gnu.utils.bugs 03/21/2002 message 9	grep.c, prline, 579
grep-color-case				grep.c, prline, 562
grep-match				grep.c, prline, 532
ping	20100214	20101006	CVE-2010-2529	ping.c, pr_options, 984
look	1.1 (svr 4)	-	37	look.c, getword, 172
indent	1.9.1	2.2.10	37	indent.c, indent, 1350

4.1 Benchmarks

Table 1 presents the loops that we use in our evaluation. The first column (Benchmark) presents the name we use to refer to the loop. The second column (Version) presents the version of the application with the infinite loop. The third column (Reference Version) presents the version of the application in which the infinite loop has been corrected. The fourth column (Bug Report) presents the source of the infinite loop bug report. The fifth column (Location) presents the file, the function, and the line number of the infinite loop.

We evaluated Jolt on eight loops in five benchmark applications. We selected applications for which 1) bug reports of infinite loops were available, 2) we could reproduce the reported infinite loops, and 3) we could qualitatively characterize the effect of escaping the loop on the application’s output. All of these applications are commonly used utilities that the user either invokes directly, from the command line, or as a part of a larger workflow:

- **ctags:** Scans program source files to produce an index that maps program entities (e.g., modules, functions, and variables) to their locations within the source files [1]. We investigate two infinite loops in ctags:
 - **ctags-*fortran*:** The ctags Fortran module (version 5.5) has an infinite loop that occurs when processing 1) source code files with variable and type declarations separated by a semicolon, or 2) syntactically invalid source files with improperly nested components. In both cases, ctags enters a mode in which it infinitely loops when it is unable to recognize certain valid Fortran keywords.
 - **ctags-*python*:** The ctags Python module (version 5.7 beta, svn commit 646) has an infinite loop that occurs when one multi-line string literal ends on a line and another multi-line string literal starts on the same line (as we discussed in Section 2.1).
- **grep:** Matches regular expressions against lines of text within a single input file or multiple input files [2]. We investigate three infinite loops in grep version 2.5. Although all of these loops are distinct, they appear to share a common origin via a copy/paste/edit development history.

- **grep-color:** This infinite loop occurs when grep is configured to display matching parts of each line in color and is given a regular expression with zero-length matches.
- **grep-color-case:** This infinite loop occurs when grep is configured to display matching parts of each line in color with case-insensitive matching and is given a regular expression with zero-length matches.
- **grep-match:** This infinite loop occurs when grep is configured to print only the parts of each line that match the regular expression and is given a regular expression with zero-length matches.
- **ping:** Ping client is a computer network utility which checks for the reachability of a remote computer using the Internet Control Message Protocol (ICMP) echo messages. The infinite loop can occur when processing certain optional headers (time stamps and trace route records) of the echo reply message from the remote computer.
- **look:** Prints all words from a dictionary that have the input word as a prefix. The infinite loop occurs when look's binary search computation visits the last entry in the dictionary file and this last entry is not terminated by a newline character. We were not able to obtain the reference version of look, but instead manually fixed the application to produce a correct result (according to our understanding of its functionality).
- **indent:** Parses and then formats C and C++ source code according to a specified style guideline [3]. This infinite loop occurs when 1) the input contains a C input preprocessor directive on the last line of the input, 2) this line contains a comment, and 3) there is no end of line character at the end of this last line.

4.2 Methodology

For each of our benchmark loops, we performed the following tasks:

- **Reproduction:** We obtained at least one input that elicits the infinite loop, typically from the bug report. Where appropriate, we constructed more inputs that cause the application to loop infinitely.
- **Loop Characterization:** We identified the conditions under which the infinite loop occurs. This includes distinctive properties of the inputs that elicit the infinite loop and characteristics of the program state. We also characterized the execution behavior (e.g., resource consumption and output) of the application during the infinite looping.
- **Infinite Loop Detection:** We first compiled the application with Jolt's compiler to produce an instrumented executable. We then ran the executable on our eliciting inputs to send the application into an infinite loop. Finally, we dynamically attached Jolt's detector to the running application to determine if Jolt could detect the infinite loop.

- **Effects of Escaping Infinite Loop:** We characterized the internal behavior of the application after using Jolt to escape the loop, including the effects of the escape on the output and the memory safety of the application. We used manual inspection and testing to ensure that the output of the application is well-formed, and Valgrind [23], to determine if the continued execution performed any invalid memory operations (such as out of bounds accesses or memory leaks).
- **Comparison with Termination:** One common strategy for dealing with an application that is in an infinite loop is to simply terminate the application. We compared the output that we obtain from terminating the application to the output from the version that uses Jolt to escape the infinite loop. Specifically, we investigated whether using Jolt helped produce a more useful output than terminating the application.
- **Comparison with Manual Fix:** We evaluated how well escaping from an infinite loop approximated the correction that the developers later made to the application. We obtained a version of the application in which the infinite loop had been manually corrected. When then compared the output from escaping the loop to the output from the fixed version of the application. Specifically, we investigated the extent to which the output produced by the application after using Jolt matched the output of the manually fixed application.

4.3 Results

Table 2 summarizes the results of our evaluation of Jolt as a technique for detecting and escaping infinite loops. The first column (Benchmark) presents the infinite loop name. The second column (Detection) presents whether Jolt successfully detected the infinite loop. If an entry in this column contains the symbol ●, detection succeeded; if it contains ○, then detection failed — we use the same notation for positive and negative results in each subsequent column.

The third column (Sanity Check) presents whether escaping the loop maintained the memory consistency, as reported by Valgrind. The fourth column (Comparison with Termination) presents whether using Jolt to escape the infinite loop produces a more useful output than the output that we obtain after terminating the application. Finally, the fifth column (Comparison with Manual Fix) presents whether using Jolt to escape the infinite loop produces the same output as the reference version for every input to the application. If an entry in this column contains the symbol ●, then the outputs are the same for some, but not all, inputs.

For infinite loops that Jolt failed to detect, we still present results that describe the behavior of the application after escaping the loop. We performed these experiments by modifying Jolt to escape the loop, even though it had not detected an infinite loop.

Table 2. Summary Results for Infinite Loops

Benchmark	Detection	Sanity Check	Comparison With	
			Termination	Manual Fix
ctags-fortran	●	●	●	◐
ctags-python	●	●	●	◐
grep-color	●	●	●	◐
grep-color-case	●	●	●	◐
grep-match	●	●	●	◐
ping	●	●	●	●
look	●	●	●	●
indent	○	●	●	●

Infinite Loop Detection: Jolt was able to detect seven out of eight infinite loops in our benchmark applications. For these infinite loops, Jolt identified that the state of the program remained the same in adjacent loop iterations, and escaped the loop immediately. Jolt failed to detect the infinite loop in `indent` because the state changed on every iteration through the loop. We discuss the reasons for why Jolt failed to detect this infinite loop in Section [5.2](#)

Sanity Check: For all of our benchmarks the resulting continued execution of the application exhibited no memory errors.

Comparison with Termination: For all our benchmarks, our evaluation indicates that using Jolt to escape the loop resulted in outputs that contain as much or more useful information than the outputs obtained by terminating the application. Terminating the applications after encountering an infinite loop left the application unable to process subsequent input units (files, lines or requests). For `ctags` and `indent` (when processing multiple input files), `grep`, `ping`, and `look`, terminating the application produced outputs only up to the point of termination (and none thereafter). `Ctags` and `indent` (when operating on a single input file with, potentially, multiple lines) are designed to produce their outputs at the end of the computation. Therefore, terminating the application did not yield any output at all. As an extreme example, terminating `indent` while in the infinite loop caused it to overwrite the input source code file with an empty file. Escaping the infinite loop with Jolt, on the other hand, not only helped the application finish processing the current input, but also enabled it to continue to successfully process subsequent inputs.

Comparison with Manual Fix: For `ping`, `look` and `indent`, the outputs of the application for which we applied Jolt, and the outputs of the application with a manually fixed bug were identical. The computations in these loops finished processing the entire input before the loop started infinitely looping.

Applying Jolt to infinite loops in `ctags` and `grep` helped produce an output containing a part of the output of the manually corrected application. In Section [2.6](#) and Section [5.1](#), we present a more detailed characterization of the quality of these outputs.

5 Selected Case Studies

In Section 2, we presented an extended case study of `ctags-python` that demonstrated Jolt's overall approach to infinite loop detection and escape. In this section we now present two additional case studies that demonstrate the main characteristics of the infinite loops that we analyzed and the details of our evaluation. In particular, these case studies highlight the utility of vigilant mode (Section 2.5), the utility of library abstraction (Section 3.2), and some of the limitations of the Jolt's detector. Detailed case studies of the rest of our benchmark applications available online at <http://groups.csail.mit.edu/pac/jolt>.

5.1 Grep

Figure 4 presents the source code of the `grep-color` loop, which colors the part of the current line matching a regular expression. Grep executes this loop when the user provides the `--color` flag on the command line. This loop, along with the other two infinite loops in `grep`, occur in the function `prline`, which is responsible for presenting the text that matches the regular expression to the user. The other two infinite loops have the same infinite loop behavior and a similar structure.

```

1  while ( (match_offset = (*execute) (beg, lim - beg, &match_size, 1))
2         != (size_t) -1)
3  {
4     char const *b = beg + match_offset;
5     /* Avoid matching the empty line at the end of the buffer. */
6     if (b == lim)
7         break;
8     fwrite (beg, sizeof (char), match_offset, stdout);
9     printf ("\33[%sm", grep_color);
10    fwrite (b, sizeof (char), match_size, stdout);
11    fputs ("\33[00m", stdout);
12    beg = b + match_size;
13 }

```

Fig. 4. Source Code for Grep-color Infinite Loop

Infinite Loop: The computation stores the pointer to the current location on the line in the variable `beg`. The function `execute()` on Line 1 searches for the next match starting from the position `beg`. Each time a match on the current line is found, this pointer is incremented to advance the search, first by adding the offset to the position of the next match (`match_offset`; Line 4), and then by adding the size of the match (`match_size`; Line 12). However, when using a regular expression that matches zero length strings (such as `[0-9]*`), the variable `match_size` will have value zero. Consequently, the value of pointer `beg` will not increase past the current match and the progress of the loop execution will stop.

The loop can still output the first non-zero length match at the beginning of the line, since `grep` uses a greedy matching strategy (it selects the longest string that matches the pattern). For example, for the input `echo "1 one" | ./grep "[0-9]*" --color`, the output contains a colored number `1`, but following loop iterations do not progress past this point — the string `one` is never printed. On the other hand, `grep-match` will output a single newline character (`'\n'`) for each iteration as it loops after the first match. In the previous example, it will output a number of newline characters after matching `1`.

Infinite Loop Detection: While in the infinite loop, the computation outputs non-printable characters (which control the text color) to the standard output stream in every iteration. The printing does not influence the termination of the loop, but may change internal counters and output buffer pointers within the standard library, which are not observable by the application, but would prevent Jolt from detecting the infinite loop. Thus, we apply library routine abstraction, which we described in Section 3.2 to allow Jolt disregard possible changes of the internal state of the library routines and enable detecting this infinite loop.

Effects of Escaping Infinite Loop: Applying Jolt to `grep` when it has entered an infinite looping state escapes the current loop (which also halts printing newline characters if the loop was doing so). The remainder of the current line is skipped. If Jolt operates in vigilant mode, `grep` will not print numerous spurious newline characters in `grep-match` case because Jolt escapes the loop after only two iterations, printing only one additional newline character. If Jolt operates in interactive mode, `grep` will print a number of newline characters for each line before the user instructs Jolt to terminate the loop causing the application to proceed to the next line.

For `grep-color` and `grep-color-case` loops, applying Jolt allows all matching lines of the input to be displayed. However, on a given line that contains multiple matches, only the first match will be colored. For example, for the sample input `echo "1 one 1" | grep -E "[0-9]*" --color`, `grep` outputs the desired line (`'1 one 1'`), but only the first `"1"` is colored. Using the `-o` command line flag to print only the matching string, `grep` outputs only the first match on each line, followed by newline characters until user invokes Jolt. For example, for the input `echo "1 one 1" | grep "[0-9]*" -o`, `grep` outputs a single line, containing a `"1"` (unlike two lines with value `"1"` that a corrected version of the application generates). Escaping the loop after printing the first match `"1"` skips the remainder of the line, which contains the second match `"1"`.

Comparison with Termination: Terminating the execution of `grep` causes it to not process any line after the first zero-length match (which is effectively any line of the input). In contrast, using Jolt allows `grep` to continue searching for matches on subsequent lines in the input.

Comparison with Manual Fix: The correction that the application developers applied for the three infinite loops in Version 2.5.3 causes the application to continue printing the line even after encountering the match of length zero. As

part of the fix, the developers completely restructured the control flow, and removed the loops in the process. This version of the application prints correctly all non-zero matches, and skips zero-length matches.

While in Section 4 we presented the results of our comparison with Version 2.5.3, we also analyzed the correction that the developers of `grep` implemented in Version 2.5.1. This fix was in place for three years before the release of 2.5.3. In this version, the developers added the code `if (match_size==0) break;` before Line 8 to exit the loop when encountering a zero-length match. The effect of this manual fix is the same as using Jolt to escape the loop.

5.2 Indent

Figure 5 presents the simplified version of the loop that handles comments that occur within or on the same line after a preprocessor directive in C programs.

```

1  while (*buf_ptr != EOL || (in_comment && !had_eof)) {
2
3      if (e_lab >= capacity_lab) e_lab = extend_lab()
4
5      *e_lab = *buf_ptr++;
6      if (buf_ptr >= buf_end) buf_ptr = fill_buffer (&had_eof);
7
8      switch (*e_lab++) {
9          case '\\':
10             handle_backslash(&e_lab, &buf_ptr, &in_comment); break;
11         case '/':
12             handle_slash(&e_lab, &buf_ptr, &in_comment); break;
13         case '"':
14         case '`':
15             handle_quote(&e_lab, &buf_ptr, &in_comment); break;
16         case '*':
17             handle_asterisk(&e_lab, &buf_ptr, &in_comment); break;
18     }
19 }
```

Fig. 5. Source Code for Indent Infinite Loop

Infinite Loop: The loop reads the text from the input buffer (pointed to by `buf_ptr`), formats it and appends it to the output buffer (pointed to by `e_lab`). The function `extend_lab()` on Line 3 increases the size of the output memory buffer if needed by using a library function `realloc()`. The function `fill_buffer()` on Line 6 reads a single line from the input file to the input buffer. If this function reads past the input file, it writes a single character `'\0'` to the input buffer and sets the `had_eof` flag. Finally, the loop body recognizes the comment's start and end characters, and sets `in_comment` appropriately.

The analysis of the loop condition on Line 1 shows that the loop computation ends only if 1) the input line contains the newline character and it is not in

the comment (`*buf_ptr == EOL && !in_comment`), or 2) if the input line contains the newline character and it has reached the end of file (`*buf_ptr == EOL && had_eof`). The loop condition does not account for the case when loop reads the entire input file, but the last line does not end with the newline character (the value of `buf_ptr` in this case is equal to `'\0'`).

Infinite Loop Detection: While in the infinite loop, each iteration appends a spurious `'\0'` character to the output buffer, and the capacity of the output buffer is occasionally increased. Eventually, the output buffer can consume all application memory and cause the application to crash. Note that this update of the output buffer is the reason Jolt in its current version cannot detect this loop as infinite.

Effects of Escaping Infinite Loop: Although Jolt cannot detect the infinite loop in this application, we manually instructed Jolt to escape the loop to investigate its effect. After escaping the loop using Jolt, the application terminated normally, producing a correctly indented output file. Note that this infinite loop only happens after `indent` has processed all of the input file; the only remaining task at this point is to copy the output buffer to a file. Escaping the loop enables the application to proceed on to correctly execute this task.

Comparison with Termination: Terminating the application when it enters the infinite loop prevents the application from executing the code to print the output buffer to the file. Because the default configuration of `indent` overwrites the input file, the user is left with an empty input file. Terminating the application also causes `indent` to skip processing any subsequent files. Escaping the loop, on the other hand, produces the correct output for the file that elicits the infinite loop, and all remaining files.

Comparison with Manual Fix: The developer fix of the loop in Version 2.2.10 modifies a condition on Line 11 to test `has_eof` flag whether the input file has reached the end, before checking for the newline character in the input buffer. Escaping the infinite loop causes the application that used Jolt to produce the same result as the reference application version.

6 Performance

In this section, we present performance measurements designed to characterize Jolt’s instrumentation overhead in normal use (Section 6.1) and the time required for detection of infinite loops in our benchmark applications (Section 6.2). We performed our performance measurement experiments on an 8-core 3.0GHz Intel Xeon X5365 with 20GB of RAM running Ubuntu 10.04.

6.1 Instrumentation Overhead

We designed this experiment to measure the overhead of adding instrumentation code to track the entry, exit, and body of each loop in the application

(as described in Section 2). We measured the overhead of Jolt’s instrumentation in normal use by running each application, with and without instrumentation, on inputs that do not cause infinite loops. In this experiment, infinite loop detection is never deployed.

Benchmarks: Our benchmark suite consists of the following workloads:

- **ctags-fortran:** we crafted five workloads by executing ctags version 5.5 with five different command line configurations on the Fortran language files of scipy, a suite of scientific libraries for the Python programming language [7]. The source code of these programs totals 81800 lines of code.
- **ctags-python:** we crafted five workloads by executing ctags version 5.7b (646) with five different command line configurations on the Python language files of numpy, a library of numerical matrix manipulation routines for Python [6]. The source code of these programs totals 72218 lines of code.
- **grep:** we crafted five workloads by executing grep version 2.5 with five different regular expressions on the concatenated C source code of grep, gstreamer [5], and sed [4]. We crafted regular expressions designed to match elements within C source code; namely, strings, comments, primitive data types (e.g., int, long, or double), parenthesized expressions, and assignment statements. The source code of these programs totals 35801 lines of code.
- **ping:** we crafted five workloads by executing ping client with different options, including targeting a remote machine on a local network (the same machine we used to reproduce the infinite loop), and the local host. We ran the same server on the remote machine that we used to elicit the infinite loop. For each ping execution we send multiple requests to the server (in particular, 100 requests to the remote host and 1,000,000 to the local host), without delay between the requests.
- **look:** we crafted five workloads by executing look version 1.1 (svr 4) with five query words and a corpus of 98569 words from the American English dictionary supplied with Ubuntu 10.04.
- **indent:** we crafted five workloads by executing indent version 1.9.1 with five different indentation styles on the C source code of gstreamer (15608 lines of code).

Methodology: To evaluate the instrumentation overhead for a single workload, we first ran the workload five times without measurement to warm the system’s file cache (and, thus, overestimate the impact of instrumentation by minimizing I/O time). We then measured the execution time of each workload twenty times across two configurations: ten times to measure the execution time of the uninstrumented application and ten times to measure the execution time of the instrumented application.

To compute the instrumentation overhead (i.e., a slowdown) for a single workload, we take the median execution time of the ten executions of the instrumented application over the median execution time of the ten executions of the uninstrumented application. We use the median to filter out executions — both slow and fast — that may be outliers due to performance variations in the execution environment.

Table 3. Performance Overhead of the Instrumentation

Benchmark	Mean	Lowest	Highest
ctags-fortran	1.073	1.068	1.080
ctags-python	1.052	1.035	1.057
grep	1.025	1.014	1.028
ping	1.016	1.005	1.024
look	1.0	1.0	1.0
indent	1.084	1.082	1.086

Results: Table 3 presents the results of the instrumentation overhead measurement experiments. The first column in Table 3 (Benchmark) presents the name of the benchmark. The second column (Mean) presents the weighted mean of the slowdowns over each benchmark’s five workloads. The third column (Lowest) presents the lowest slowdown that we observed over each benchmark’s five workloads. The fourth column (Highest) presents the highest slowdown that we observed over each benchmark’s five workloads.

Jolt’s overhead varies between 0.5% (the lowest observed overhead for ping) and 8.6% (the highest observed overhead for indent). In our experiments we found that the overhead imposed by Jolt on look was, in practice, too small to reliably distinguish it from the noise of the benchmark environment. We also note that the results for ping depend on the status of the network and the physical distance between the hosts. While we used short physical distances between the hosts and no delay between the requests to decrease the network variability and to account for the worst case, we expect that in a typical use the communication time will dominate the processing time, making the overhead negligible.

6.2 Infinite Loop Detection

We designed this experiment to evaluate how quickly Jolt can detect an infinite loop in a running application.

Methodology: To perform this experiment, we ran each infinite loop from our case studies on an input that elicits an infinite loop and then attached Jolt. We then allowed Jolt to run for two seconds; if Jolt did not detect the loop within two seconds, then we classified the loop as undetectable.

For each detected infinite loop in our case studies, we gathered 1) the time required for Jolt to detect the infinite loop, 2) the footprint, in number of bytes, of each infinite loop iteration, and 3) the length, in number of instructions, of each infinite loop. Each of these numbers corresponds to the second, third, and fourth columns of Table 4, respectively:

- **Time:** To measure the detection time, we repeatedly (five times) measured the absolute time that elapsed from the instant when Jolt attached to the application until the instant when Jolt detected that the loop iterations do not change the state. We report the median detection time over these trials.

Table 4. Infinite Loop Detection Statistics for Benchmark Applications

Benchmark	Time (s)	Footprint (bytes)	Length
ctags-fortran	0.319	240	256
ctags-python	0.334	312	992
grep-color	0.585	992	4030
grep-color-case	0.579	992	4036
grep-match	0.490	846	2506
ping	0.287	192	54
look	0.296	300	378

- **Footprint:** We measured the memory footprint of the infinite loop by recording the number of bytes of the program state that Jolt recorded in the snapshot at the beginning of each iteration of the infinite loop. As discussed in Section 3, Jolt records the value of a register or memory address at the beginning of the loop only if it was written during the execution of the loop.
- **Length:** We measured the length of the loop by recording the number of instructions dynamically executed during one iteration of the loop. Our reported numbers count only user-mode instructions that wrote to a register or a memory location — this, therefore, excludes instructions executed by the operating system kernel and nop instructions that do not modify the state of the application.

Results: Table 4 presents the results of our infinite loop detection experiment. The times required to detect an infinite loop are all less than 1 second and the footprint of each infinite loop is less than 1 KB. Given that ping’s infinite loop, our smallest benchmark loop (by number of instructions), takes Jolt 0.287 seconds to detect, our infinite loop detection technique is predominantly bounded from below by the time required to initialize the Pin instrumentation framework.

7 Limitations

Jolt currently detects only infinite loops that do not change program state between iterations. In general, infinite loops can change state between iterations, or cycle between multiple recurring states. We anticipate that Jolt’s detector could be extended to eliminate changing state that does not affect a loop’s termination condition, track multiple states, or use symbolic reasoning to prove non-termination [14, 35, 9].

Jolt does not consider the effects of multiple threads on the termination of the application. For example, Jolt may incorrectly report that an application is in an infinite loop if the application uses ad-hoc synchronization primitives (e.g., spin loops) [38]. In our evaluation, we only considered single-threaded applications.

Jolt does no further intervention after allowing the application to escape an infinite loop. In principle, it is possible for an application to escape an infinite loop and then crash, producing no output. In our evaluation, we inspected the

source code of our applications to determine that they continue their execution without crashing, and eventually produce outputs. We anticipate that Jolt could be extended to use any of a number of program shepherding techniques to help steer programs around potential errors [28,13,24,30].

8 Related Work

Researchers have previously studied the causes for program failures, including unresponsiveness, in operating systems, server applications, and web browsers [20,17,32,25]. In particular, Song et al. identify infinite loops as an important cause of unresponsiveness in three commonly used server applications and in a web browser. This paper identifies the causes of eight infinite loops in existing utility applications. Our evaluation also shows that seven of these loops can be detected by checking that their state does not change across loop iterations.

(Non-) Termination Analysis: Researchers have previously suggested using program analysis to identify infinite loops during software development. Gupta et al. [14] present TNT, a non-termination checker for C programs, which identifies infinite loops by checking for the presence of recurrent state sets, which are sets of program states that cause a loop to execute infinitely. TNT uses template-based constraint satisfaction to identify sets of linear inequalities on program variables that describe recurrent state sets. Velroyen et al. [35] also propose a template-based constraint invariant satisfaction approach to identify infinite loops — though with a different invariant generation technique. Burnim et al. developed Looper, a tool that uses symbolic execution and Satisfiability Modulo Theories (SMT) solvers to infer and prove loop non-termination arguments [9].

Each of these approaches could, in principle, be used to attach to a running instance of a program and detect an infinite loop. And, in fact, developers can use Looper [9] to break into a debugging mode to prove that a suspect loop is infinitely looping. While these approaches can identify a larger class of infinite loops than Jolt — i.e., infinite loops that change state on each iteration — this power comes at the cost of symbolic execution, SAT solving, or SMT solving. Jolt, in contrast, attaches to the concrete execution of the program and uses an inexpensive detection mechanism to identify infinite loops that do not change state. In addition, Jolt provides users with the option to escape detected infinite loops and continue the execution of the program.

Researchers have also developed static analysis tools that can be used during program development to determine statically, when possible, whether each loop in the program terminates [11,10,8,33]. We view these approaches as complementary in that it would be possible to incorporate the results of static analysis into Jolt’s instrumentation decisions. Namely, if it can be proven statically that a particular loop will terminate, then Jolt need not instrument that loop.

Program Repair: Nguyen and Rinard have previously deployed an infinite loop escape algorithm that is designed to eliminate infinite loops in programs that use cyclic memory allocation to eliminate memory leaks [24]. The proposed

technique records the maximum number of iterations for each loop on training inputs, and uses these numbers to calculate a bound on the number of iterations that the loop executes for previously unseen inputs. To the best of our knowledge, this is the only previously proposed technique for automatically escaping infinite loops. In comparison to the approach we present in this paper, Nguyen and Rinard’s technique is completely automated, but may also escape loops that would otherwise terminate.

Researchers have also investigated techniques for general program repair that could, in principle, automatically generate fixes for infinite loops [31, 27, 30, 26, 12, 36, 29]. Weimer et al. [37] have used genetic programming to automatically generate program repairs from snippets of code that already exist in the program. In their evaluation, they used their technique to generate fixes for the infinite loops in look and indent, which we also used in our evaluation. Their automatically generated fixes eliminated the infinite loops, but at the cost of some lost functionality of the application. Compared to these fixes, escaping an infinite loop enables a user to recover the complete outputs of these applications.

Handling Unresponsive Programs: Finally, we note that operating systems and browsers often contain task management features that allow users to terminate unresponsive or long-running applications or scripts. Mac OS X, for example, provides a Force Quit Applications user interface; Windows XP provides a Windows Task Manager. Web browsers also contain user interface features that alert users to long-running scripts and offer users the option of terminating these scripts [19]. However, these facilities usually offer only termination of a long-running task, while Jolt allows for the potential continued execution after the long-running loop subcomputation. Extending Jolt to work in these environments would provide the user with the additional option of detecting and escaping infinite loops in unresponsive or long-running applications.

9 Conclusion

By making applications unresponsive, infinite loops can cause users to lose work or fail to obtain desired output. We have implemented and evaluated Jolt, a system that detects and, if so instructed, escapes infinite loops. Our results show that Jolt can enable applications to transcend otherwise fatal infinite loops and continue on to produce useful output. Jolt can therefore provide a useful option for users who would otherwise simply terminate the computation.

Acknowledgements. We would like to thank Professor Armando Solar-Lezama for providing the motivating example presented in the Introduction. We would also like to thank Hank Hoffmann, Deokhwan Kim, Fan Long, Stelios Sidiroglou, Karen Zee and the anonymous reviewers for their useful comments on the earlier drafts of this paper. We would also like to thank Claire Le Goues for providing us with the bug-containing versions of look and indent benchmarks.

This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241 and IIS-0835652) and the United States Department of Energy (Grant DE-SC0005288).

References

1. Exuberant ctags, <http://ctags.sourceforge.net>
2. GNU grep, <http://www.gnu.org/software/grep>
3. GNU indent, <http://www.gnu.org/software/indent>
4. GNU sed, <http://www.gnu.org/software/sed>
5. GStreamer, <http://www.gstreamer.net>
6. numpy, <http://numpy.scipy.org>
7. scipy, <http://www.scipy.org>
8. Bradley, A.R., Manna, Z., Sipma, H.B.: Termination of polynomial programs. In: International Conference on Verification, Model Checking, and Abstract Interpretation (2005)
9. Burnim, J., Jalbert, N., Stergiou, C., Sen, K.: Looper: Lightweight detection of infinite loops at runtime. In: International Conference on Automated Software Engineering (2009)
10. Colón, M.A., Sipma, H.B.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
11. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: beyond safety. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 415–418. Springer, Heidelberg (2006)
12. Dallmeier, V., Zeller, A., Meyer, B.: Generating fixes from object behavior anomalies. In: International Conference on Automated Software Engineering (2009)
13. Demsky, B., Rinard, M.: Data structure repair using goal-directed reasoning. In: International Conference on Software Engineering (2005)
14. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: Symposium on Principles of Programming Languages (2008)
15. Hayashizaki, H., Wu, P., Inoue, H., Serrano, M., Nakatani, T.: Improving the performance of trace-based systems by false loop filtering. In: International Conference on Architectural Support for Programming Languages and Operating Systems (2011)
16. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International Symposium on Code Generation and Optimization (2004)
17. Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C.: Have things changed now? an empirical study of bug characteristics in modern open source software. In: Workshop on Architectural and System Support for Improving Software Dependability (2006)
18. Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: Conference on Programming Language Design and Implementation (2005)
19. Metz, C.: Mozilla girds firefox with hang detector (June 2010), http://www.theregister.co.uk/2010/06/10/firefox_hang_detector/
20. Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A., Steidl, J.: Fuzz revisited: A re-examination of the reliability of Unix utilities and services. TR #1268, Computer Sciences Department, University of Wisconsin (1995)
21. Muchnick, S.: Advanced Compiler Design and Implementation (1997)
22. Nagarajan, V., Jeffrey, D., Gupta, R.: Self-recovery in server programs. In: International Symposium on Memory Management (2009)

23. Nethercote, N., Seward, J.: Valgrind A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science* 89(2), 44–66 (2003)
24. Nguyen, H.H., Rinard, M.: Detecting and eliminating memory leaks using cyclic memory allocation. In: *International Symposium on Memory Management* (2007)
25. Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J., Muller, G.: Faults in Linux: Ten years later. In: *International Conference on Architectural Support for Programming Languages and Operating Systems* (2011)
26. Perkins, J., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W., Zibin, Y., Ernst, M., Rinard, M.: Automatically patching errors in deployed software. In: *Symposium on Operating Systems Principles* (2009)
27. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: treating bugs as allergies—a safe method to survive software failures. In: *Symposium on Operating Systems Principles* (2005)
28. Rinard, M., Cadar, C., Dumitran, D., Roy, D.M., Leu, T., Beebe Jr., W.S.: Enhancing server availability and security through failure-oblivious computing. In: *Symposium on Operating Systems Design and Implementation* (2004)
29. Schulte, E., Forrest, S., Weimer, W.: Automated program repair through the evolution of assembly code. In: *International Conference on Automated Software Engineering* (2010)
30. Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., Keromytis, A.: Assure: automatic software self-healing using rescue points. In: *International Conference on Architectural Support for Programming Languages and Operating Systems* (2009)
31. Sidiroglou, S., Locasto, M., Boyd, S., Keromytis, A.: Building a reactive immune system for software services. In: *USENIX Annual Technical Conference* (2005)
32. Song, X., Chen, H., Zang, B.: Why software hangs and what can be done with it. In: *International Conference on Dependable Systems and Networks* (2010)
33. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for java bytecode based on path-length. *Transactions on Programming Languages and Systems* 32, 1–70 (2010)
34. Theiling, H.: Extracting safe and precise control flow from binaries. In: *International Conference on Real-Time Systems and Applications* (2000)
35. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In: *International Conference on Tests and Proofs* (2008)
36. Wei, Y., Pei, Y., Furia, C., Silva, L., Buchholz, S., Meyer, B., Zeller, A.: Automated fixing of programs with contracts. In: *International Symposium on Software Testing and Analysis* (2010)
37. Weimer, W., Nguyen, T.V., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: *International Conference on Software Engineering* (2009)
38. Xiong, W., Park, S., Zhang, J., Zhou, Y., Ma, Z.: Ad hoc synchronization considered harmful. In: *Symposium on Operating Systems Design and Implementation* (2010)

Atomic Boxes: Coordinated Exception Handling with Transactional Memory

Derin Harmanci¹, Vincent Gramoli^{1,2}, and Pascal Felber¹

¹ University of Neuchâtel, Switzerland

² EPFL, Switzerland

Abstract. In concurrent programs raising an exception in one thread does not prevent others from operating on an inconsistent shared state. Instead, exceptions should ideally be handled in coordination by all the threads that are affected by their cause.

In this paper, we propose a Java language extension for coordinated exception handling where a named `abox` (atomic box) is used to demarcate a region of code that must execute *atomically* and in isolation. Upon an exception raised inside an `abox`, threads executing in dependent `aboxes`, roll back their changes, and execute their recovery handler in coordination. We provide a dedicated compiler framework, CXH, to evaluate experimentally our atomic box construct. Our evaluation indicates that, in addition to enabling recovery, an atomic box executes a reasonably small region of code twice as fast as when using a failbox, the existing coordination alternative that has no recovery support.

Keywords: error recovery, concurrent programs, failure atomicity.

1 Introduction

Exceptions and exception handling mechanisms are effective means for redirecting the control flow of an error-prone sequential program before it executes on an inconsistent state of the system. This fact has led to extensive studies on exception handling mechanisms and their being tailored to work well with sequential programs. At the same time, a recent survey on 32 sequential applications presents the general picture on the exception handling usage by the programmers and reports that even though more than 4% of the total source code is dedicated to it, exception handling is still neglected in most of the cases: either terminating the program or ignoring the exception [1]. This result shows that sequential programs are generally developed by using exceptions as a means to terminate programs in a convenient way and inconsistencies resulting from exceptional situations are not really treated.

In concurrent programs, however, an exception raised by one thread cannot prevent other threads from accessing an inconsistent shared state because other threads may not be aware of the raised exception. Such an exception should ideally be detected by all the threads that operate on the same shared state because they can be affected by its cause. Two solutions to the problem can

be considered: (i) the program should be brought back to a consistent state by handling the exception, or (ii) all the affected threads (or even the whole application) should be terminated to avoid execution on inconsistent shared states. Since there are no widespread mechanisms that allow any of these solutions, it is up to programmers to devise a solution for such cases. In other words, compared to sequential programs where treating exceptions is barely considered, for concurrent programs handling exceptions should be part of the main application design and development in order not to jeopardize the application correctness.

To illustrate how easily the above inconsistency problem can appear in ordinary concurrent programs, consider the following code in Figure 1 (inspired by a similar example in [2]). The figure presents a naive implementation of a classifier program where multiple threads concurrently evaluate nodes from the `unclassifiedNodes` list, process them, and move them to the target class using the `assignToClass` method. Note that we assume that both the `unclassifiedNodes` list and the target classes `class[N]` are shared by all threads.

```

1 Class NodeClassifier {
2   int N; // number of classes
3   List unclassifiedNodes;
4   Set class[N];
5   ...
6   public void assignToClass(int srcPos, int targetClass) {
7     synchronized(this) {
8       Node selectedNode = unclassifiedNodes.remove(srcPos);
9       selectedNode.transform();
10      class[targetClass].add(selectedNode);
11    }
12  }
13 }
```

Fig. 1. A concurrent code that may end up in an inconsistent state if an exception is raised while the selected node's representation is being transformed as required by the target class object in `selectedNode.transform()`

When an exception is raised on line 9, the system reaches an inconsistent shared state if the exception is not handled: the `selectedNode` gets lost as it is neither in the `unclassifiedNodes` nor in its target class. For correct execution of the program, the exception should be handled and this should be performed before any of the other threads, unaware of the raised exception, access either the `unclassifiedNodes` list or the target class, which are inconsistent. Hence, the handling of the exception should take the existence of concurrent threads into account.

This example, albeit naive, clearly shows that exception handling becomes a first class design consideration in development of correct concurrent programs. And, needless to say, with the mainstream computer hardware becoming multi-core, concurrent programming is about to become mainstream too. This fact

highlights the need for solutions that will simplify concurrent programming under exceptional situations.

Recently, Jacobs and Piessens proposed *failbox* as a mechanism to prevent the system from running in such an inconsistent state. The key idea is that, if one thread raises an exception in a failbox, any other thread is prevented from executing in the same failbox [3]. Instead of letting the system run in an inconsistent state, a failbox simply halts all concurrent threads accessing the same failbox. However, failboxes neither revert the system to a consistent state nor help the programmer recover from the error.

In this paper, we propose an **abox-recover** construct as a language extension that supports coordinated exception handling by providing **abox** and **recover** blocks (the keyword **abox** is derived from “atomic box”). Our **abox-recover** construct differs radically from the failbox extension, as it reverts the system to a consistent state upon exception to enable recovery through coordinated exception handling. Hence, **aboxes** do not only propagate the exceptions to concurrent threads of the system (as failboxes do), but also allow these threads to recover from this exception in a coordinated manner.

The programmer uses a named **abox** to demarcate regions of code that should remain in a consistent state upon exception. The **abox** guarantees failure atomicity either by executing all its content or by reverting its modifications. Failure atomicity allows the programmer to handle exceptions. For example, by replacing **synchronized** with **abox** in Figure 1, the inconsistency problem can be solved: the **abox** reverts all its changes including the modification of `unclassifiedNodes`. Dependencies between **aboxes** are indicated using a simple naming scheme: *dependent* **aboxes** (ones that are subject to inconsistencies related to the same data) are attributed the same name. If an exception is raised inside an **abox**, all threads executing in dependent **aboxes** stop their execution and rollback their changes. Then, execution continues in a **recover** block, analogous to a **catch** block, by one or all the affected threads as specified by the programmer. Typically, the **recover** block aims at correcting the condition that caused the exception and/or reconfigure the system before redirecting the control flow, restarting for example the execution of the atomic box. Our **abox-recover** construct therefore provides the simplicity of a **try-catch**, but for coordinated exception handling in multi-threaded applications.

Contributions. We propose an **abox-recover** construct as a language extension for coordinated exception handling. Our **abox** block uses *transactional memory* (TM), a concurrent programming paradigm ensuring that sequences of memory accesses, *transactions*, execute atomically [4]. As far as we know, **abox** is the first language construct that benefits from memory transactions for concurrent exception handling.

More specifically, an **abox** acts like a transaction that either *commits* (all its changes take effect in memory), or *aborts* (all its effects are rolled back). The main difference between **abox** and memory transactions lies in the way commit and abort are triggered. In TM, transactions abort only if a detected conflict prevents the transaction from being serialized with respect to concurrently

executing transactions. With coordinated exception handling, an `abox` is rolled back also if an exception has been raised inside a dependent `abox`, which leads to the execution of the corresponding `recover` block.

We have implemented a compiler framework for coordinated exception handling, called CXH, that converts `aboxes` into some form of transactions. Our CXH compiler framework ensures that all `aboxes` execute speculatively, making sure that no exceptions are raised before applying the changes of the corresponding `aboxes` in the shared memory. More precisely, CXH consists of a dedicated Java pre-compiler that converts our language extensions into annotated Java code, which is executed using a TM library thanks to an existing bytecode instrumentation agent. The CXH compiler generates code that guarantees that, if an exception is raised in an `abox`, each thread executing a dependent `abox` concurrently gets notified and rolls back the changes executed in the corresponding `abox`. Depending on the associated `recover` block, the threads perform appropriate recovery actions and restart or give up the execution of the `abox`.

We compare experimentally our `abox-recover` construct against failboxes, which only stop threads running in the same failbox without rolling back state changes. Our results indicate that `aboxes` that comprise up to few hundreds memory accesses execute $2\times$ faster than failboxes in normal executions, where no exceptions are raised, and $15\times$ faster than failboxes to handle exceptions. We also tested extreme cases where an `abox` executes thousands of memory accesses, in which case the cumulated overhead of TM accesses may result in lower performance than long failboxes. Besides illustrating that TM is a promising paradigm for failure atomicity and strong exception safety, our evaluation indicates that the `abox` mechanism is efficient compared to similar techniques providing weaker guarantees.

Roadmap. Section 2 presents the background and related work. Section 3 introduces an example that is used subsequently to illustrate our language constructs. Section 4 describes the syntax and semantics of the language constructs for coordinated exception handling. Section 5 presents the implementation of coordinated exception handling and our CXH compiler framework. Section 6 compares the performance we obtained against failboxes and Section 7 concludes.

2 Background and Related Work

Concurrent recovery. Thanks to their ability to rollback and their isolation from the other parts of the program, atomic transactions have been used for concurrent handling of exceptions since the eighties [5]. Transactions by themselves have been considered useful only for *competitive concurrency* where concurrently executing threads execute separately, unaware of each other, but access common resources. This type of concurrency is the primary target of our approach.

A classical alternative to avoid inconsistencies in portions of programs generating competitive concurrency consists in encapsulating the associated code in transactions. Argus [6], Venari/ML [7] and Transactional Drago [8] map transactions to methods (within which multiple threads can be spawned) and allow

an exception that cannot be resolved on a local thread to abort the transaction, passing the exception to the context where the method is called. OMTT [9] allows existing threads to join a transaction but still propagate exceptions to a context outside the transaction. In these approaches, exceptions concerning all the competing threads result in the rollback of the transaction and the propagation of the exception outside the transaction. In contrast, our approach allows threads to (cooperatively) handle such exceptions, instead of directly propagating the exception outside of the transaction scope.

The secondary target of our approach is *cooperative concurrency* that occurs when multiple threads communicate to perform a common task. The mainstream solution for cooperative concurrency is coordinated atomic (CA) actions that propose to complement transactions with conversations to provide coordinated error recovery. This approach applies to distributed objects like clients and databases in a message passing context [10], e.g., the systems surveyed in [11] whose distributed modules are presented in [12]. In contrast, our approach targets modern multi-core architectures thus benefiting from shared memory to coordinate efficiently the recovery among concurrent threads. For example, our approach shares the concept of guarded isolated regions for multi-party interactions from [13] without requiring a manager to synchronize the distributed interaction participants. Furthermore, a programmer needs to include a significant amount of code to construct the CA action structure in her program using frameworks specifically designed for this purpose [11,14], whereas in our approach the programmer can simply relate dependent code regions of an atomic box using built-in language constructs and their parameters.

A more recent checkpointing abstraction for concurrent ML, called *stabilizers*, monitors message receipts and memory updates to help recovering from errors [15]. Stabilizers can cope with transient errors but do not allow coordinated exception handlers to encompass permanent errors.

Failboxes [3] ensure cooperative detection of exceptions in Java. A thread that raises an exception while executing the code encapsulated in a failbox sends a signal to the concurrent threads that are also executing in the same failbox. Upon reception of this signal an exception is raised so that all threads can terminate, which ensures that no thread keeps running on a possible inconsistent shared state. Failbox does not provide coordinated exception handling because the inconsistent state produced by the error cannot be reverted, hence the system has no other solutions but stopping. One could use failboxes to stop the entire concurrent program and restart it manually, however, restarting the program from the beginning may not prevent the same exception to occur again. In contrast, **aboxes** automatically rollback their changes upon exceptions and let the programmer define recovery handlers to remedy the cause of an exception and redirect the control flow.

Transactional memory. Transactional memory (TM) [4] is a concurrent programming paradigm that lets the programmer delimit regions of code corresponding to *transactions*. A TM ensures that each transaction appears to be executing atomically: either it is aborted and none of its changes are visible

from the rest of the system, in which case the transaction can be restarted, or it commits and writes all its changes into the shared memory. The TM infrastructure checks whether memory locations have been accessed by concurrent transactions in such a way that conflicts prevent them from being serialized, i.e., from being executed as if they were sequentially ordered one after the other. In such case, one of the conflicting transactions has to abort.

The inherent isolation of transactions may seem a limitation to achieve high levels of concurrency with some cooperative programming patterns, such as producer-consumer interactions that have inter-thread dependencies. Several contention management policies for TMs have been proposed, however, to alleviate this problem and provide progress guarantees [16,17]. Indeed a TM conveys a simple rollback mechanism on which one can build coordinated exception handling. While originally proposed in hardware [18], many software implementations of TMs have since been proposed [19,20,21,22,23,24,25].

More recently, transactional (atomic) blocks have been suggested as a potential solution for exception handling. Shinnar et al. [26] proposed a `try_all` block for C#, which is basically a `try` block capable of undoing the actions performed inside the block. Cabral and Marques [27] similarly propose to augment the `try` block with transactional semantics (using TM as underlying mechanism) to allow the retry of a `try` block when necessary. Other work proposed richer atomic block constructs that build upon TM and that help with exception handling [28,29,30]. However, all the existing implementations for the above work focus on sequential executions, hence being unable to cope with coordinated exception handling. When a thread raises an exception, it can either rollback or propagate the exception. If the exception is not caught correctly, the thread may stop and leave the memory in a corrupted state that other threads may access.

3 A Running Example

In this section, we introduce an example code (Figure 2) which we later use to explain different aspects of atomic boxes. The example represents a multi-threaded application with a shared task queue `taskQueue` from which threads get tasks to process. All threads execute the same code. Once a thread obtains a task, it first performs pre-computation work (getting necessary inputs and configuring the task accordingly) in the `prepare` method. The execution of the task is performed in the `execute` method of the thread, by calling sequentially the `process` and `generateOutput` methods of the task. We assume that `generateOutput` can add new tasks in the `taskQueue`.

In what follows, we will mainly focus on the `execute` method of the thread. The code of the method is given without any exception handling. The traditional approach would be to use a `try-catch` statement enclosing the content of the `execute` method. However, when an exception is caught, one cannot easily determine at what point the execution of the method was interrupted and hence, in general, it is difficult to revert to the state at the beginning of the method. In such a case the `task` object could stay in an inconsistent state, possibly even

```

1  public void run() {
2      Task task = null;
3      while(true) {
4          synchronized(taskQueue) {
5              task = taskQueue.remove();
6          }
7          if (task == null) break;
8          prepare(task);
9          execute(task);
10     }
11 }
12
13 public void prepare(Task task) {
14     task.getInput();
15     task.configure();
16 }
17
18 // No exception handling
19 public void execute(Task task) {
20     task.process();
21     task.generateOutput();
22 }

```

Fig. 2. A simple example where multiple threads process tasks from a common task queue and that would benefit from concurrent exception handling

affecting the state shared with other threads, and it would not be possible to simply put the task back into the `taskQueue` for later re-processing. The loss of a task might require other threads to reconfigure, or to stop execution altogether for safety or performance reasons: shared state may be inconsistent, incomplete processing would be worthless. We will see in the next section using this example how atomic boxes prevent the loss of the task and how they allow us to correct the cause of the exception and coordinate threads for the program to recover.

4 Syntax and Semantics

Our language extension deals mainly with code blocks that are dependent on each other in the sense that if a statement in one of the blocks raises an exception not handled within the block, none of the other code blocks should continue executing. We call such blocks *dependent blocks*. An atomic box is a group of dependent code blocks that are dependent and can act together to recover from an exception that is raised in at least one of the code blocks. In order to express an atomic box, each dependent code block belonging to an atomic block is enclosed inside a new Java statement, `abox-recover`. The fact that `abox-recover` statements belong to the same atomic box is specified by assigning them the same name. The name of an atomic box is assigned to an `abox-recover` statement as a parameter.

An atomic box can be descendant of another atomic box, which means that the atomic box is dependent on the parent atomic box. Relating an atomic box as a descendant of another atomic box is achieved by assigning a descendant name in the hierarchical naming space. If associated `abox-recover` statements of the same atomic box execute on different threads, these threads are said to be executing in the same atomic box.

Basically, an `abox-recover` statement is composed of two consecutive blocks: the first block is called `abox` and the second `recover`. The precise syntax of the `abox-recover` statement can be described as follows:

```

abox [ ("name", <handlingContext> ) ]
  { S }
[ recover(ABoxException <exceptionName>)
  { S' } ]

```

where `abox` and `recover` are keywords, *S* and *S'* are sequences of statements (that may include the additional keywords `retry` and `leave` introduced by our language extension), `name` and `<handlingContext>` are parameters of the associated `abox` keyword and the `<exceptionName>` is the parameter of the `recover` keyword. Optional parameters and structures are enclosed in square brackets: `abox` may have no parameter and the block `recover` is optional.

An `abox` encloses a dependent code block of the application, while the `recover` block specifies how exceptions not caught in the `abox` are handled. If an unhandled exception is raised in an `abox`, we say that the `abox` fails. An `abox-recover` statement provides the convenience of `try-catch` to a dependent block with the following notable differences:

- *Failure atomicity*: An `abox` of an `abox-recover` statement can be rolled back, i.e., either the contents of the `abox` performs all of its modifications successfully (thus none of the `aboxes` that belong to the same atomic box fail at any point), or the `abox` acts as if it has not performed any modifications. The failure atomicity property of the `abox` is possible because an `abox` is executed inside a *transaction*.
- *Dependency-safety*: An atomic box ensures dependency safety; i.e., if a statement fails raising an exception, all statements that depend on the failing statement do not execute. The dependency relation between statements is established by naming `abox-recover` statements with a common name (or with names of descendants). The dependency-safety is ensured by two properties of `abox-recover` statement: *i*) An `abox` executes in a transaction, thus its execution is isolated from all dependent code in the system until it commits. In other words, none of the dependent code blocks see the effects of each other as long as code blocks do not commit. *ii*) If an exception is not handled in an `abox` it rolls back its changes and recovery actions are taken only after all the `aboxes` of an atomic box are rolled back. Thus, in no situation it is possible for a dependent code block to see partial modifications of the another dependent block that is in inconsistent state.
- *Coordinated exception handling*: A `try-catch` statement offers a recovery from exception only for the thread on which the exception occurs. The

abox-recover statement allows the programmer to inform concurrently executing threads of an exception raised in one of the threads. Moreover, through the **recover** block of the **abox-recover** statement it is possible to recover from that exception in a coordinated manner. Note that the coordination is possible among **recover** blocks because they do not execute in a transaction.

- Last, an **abox** and its associated **recover** block can include **try-catch** statements to handle exceptions raised in their context.

We distinguish two different modes of operation for an **abox-recover** statement: *normal mode* and *failure mode*. The normal mode is associated with **abox** and the failure mode is associated with the **recover** block. An **abox** executes in normal mode, i.e., an **abox** executes as long as no exceptions are raised or until an exception raised inside **abox** propagates outside of the block. Note that if the code inside **abox** raises an exception, and this exception is caught in the block itself, the **abox** still executes in normal mode.

When an exception is propagated out of **abox** boundaries (i.e., when an unhandled exception is raised in the **abox**), the **abox** is said to fail and its **abox-recover** statement switches to failure mode. The failure model of the **abox-recover** statement is such that when the block **abox** fails, its associated atomic box also fails (because the atomic box acts as a single entity upon an exception). Thus, all the **abox-recover** statements associated to the atomic box switch to failure mode upon the failure of an **abox**. The failure of an **abox** also triggers the failure of the descendent atomic boxes.

In the failure mode all the threads that execute in the atomic box coordinate together. They wait for each other to ensure that all the associated **abox-recover** statements switch to failure mode and all the **aboxes** are rolled back. Then they perform recovery actions as specified by the **abox** where the exception is raised. After the recovery actions are terminated all the threads decide locally how to redirect their local control flow. There are three options in redirecting the control flow at the end of recovery: restarting, continuing with the statement that comes after **abox-recover** statement, or raising an exception (i.e., abrupt termination). The first two options are provided through two new control flow keywords (**retry** and **leave** respectively), while raising an exception is done by the usual **throw** statement.

In the rest of this section, we detail the constructs for normal and failure modes, the control flow keywords, and the nesting of atomic boxes. We will also discuss the semantics of the **abox-recover** statement under concurrently raised exceptions.

4.1 Normal Mode Constructs

The only normal mode construct introduced by our language extension is the **abox**. An **abox** encloses a dependent code block of an atomic box. The block is part of the application code and the fact that it is enclosed in an **abox** does not modify its functionality except for exception handling. In other words, as long as no exception is propagated out of the dependent code block, there is no

difference in terms of correctness of the application to have the block in an `abox` or not. However, inserting the code in an `abox` increases safety and provides a means for handling exceptions across multiple threads.

Although the functionality of the code inserted in an `abox` is not modified, an `abox` has different semantics compared to traditional blocks: `abox` executes as a transaction. That way, the modifications performed by the code inside the `abox` are only guaranteed to be effective if the `abox` successfully terminates (hence, if it successfully commits without switching to failure mode). Otherwise none of the modifications performed in the context of the `abox` are visible by code outside the `abox`. Therefore, the code in an `abox` executes atomically and in isolation.

The transactional nature of the `abox` normal execution does not have effect on the correctness of enclosed code but has implications on its execution time. As the transactional execution is provided by an underlying transactional memory (TM) runtime, it incurs two types of latency overhead: *i*) data accesses in the `abox` are under the control of TM and will be slower than bare data accesses; *ii*) in multi-threaded code if different `aboxes` concurrently perform accesses on shared data in a way that inconsistencies would occur, an `abox` may be aborted, rolled back and restarted, which adds extra latency to its execution.

In its simplest form (i.e., when its optional parameters are omitted) the syntax for an `abox` is

```
abox { S }
```

where S is a sequences of statements. The statements in S may contain traditional Java statements as well as the control flow keywords added by our language extension (see Section 4.3). For the sake of simplicity, in this paper we do not consider Java statements that perform irrevocable actions (e.g., I/O operation or system calls) in an `abox` because most underlying TM implementations do not support transactional execution for such actions. There exist however practical solutions to this limitation (e.g., in [31,32]).

The simplest form of an `abox` is considered as an indication that the block is the only block in an atomic box, and thus it does not have any dependencies on other parts of the code. For such `abox` the exception handling is done locally without any coordination with any other `abox`. Hence, this form is suitable for exception handling in single-threaded applications as well as handling of exceptions for code blocks of multi-threaded applications that do not have any implications on other running threads.

As an example of such scenario, assume that an `OutOfMemoryError` is raised during the execution of the `execute` method of Figure 2. If for the running multi-threaded application, it is known that most of the tasks has small memory footprint but occasionally some tasks can have large memory footprint (but never exceeding the heap size allocated by the JVM), it is possible to clean up some resources or wait for a while before restarting execution. This would solve the problem as memory is freed when a task with a possibly large footprint finishes executing. Using the simple form of `abox`, the code for this solution would be as in Figure 3 (the syntax for the `recover` block will be explained shortly). Note


```

1  public void execute(Task task) {
2      abox {
3          task.process();
4          task.generateOutput();
5      } recover(ABoxException e) {
6          if(e.getCauseClass() == OutOfMemoryError.getClass()) {
7              // Back off (sleep) upon OutOfMemoryError
8              backOff();
9          }
10         // Implicit restart
11     }
12 }

```

Fig. 3. Local recovery for an `OutOfMemoryError` using the simple form of `abox`

that this solution is not possible with either a `try-catch` block or a failbox since the state of the `task` object cannot be rolled back to its initial state.

A programmer can describe an atomic box composed of multiple `aboxes` by assigning all of the associated `aboxes` the same name. The syntax for expressing an `abox` of such an atomic box is:

```

abox("name", <handlingContext>) { S }

```

where the `name` parameter is a string that associates the `abox` to the atomic box it belongs to, and the `<handlingContext>` parameter is a keyword describing which `recover` blocks will execute for performing recovery. Since the `<handlingContext>` parameter effects the execution of `recover` block, details on this parameter are provided with the description of `recover` block in Section 4.2.

Contrarily to the simplest form of `abox`, the named form implies that upon failure of the `abox` the exception handling should be coordinated across the atomic box. This form serves mostly in handling exceptions in multi-threaded applications.

We can slightly change the conditions to the example for which `abox` provided a solution in Figure 3 and generate a different scenario. Let us assume that in the example there are not many solutions for solving the `OutOfMemoryError` and the programmer simply wants to stop all the threads when such an exception is raised. The code that will provide this solution would be as in Figure 4.

Note that all the threads are running the same code. The code in Figure 4 uses the named form of `abox`. The `<handlingContext>` parameter is given as `all`, which means that when the `OutOfMemoryError` is raised on one thread, all the threads running in the atomic box will execute their `recover` blocks. In the `recover` block an exception is raised so that the currently executing thread dies (since the threads are assumed to be running the code in Figure 2, the exception will not be caught and each thread will be terminated). This solution is again not possible with a `try-catch` statement. Since the objective in this example is to stop the application, the failbox approach would also work: one could enclose

```

1  public void execute(Task task) {
2      abox("killAll", all) {
3          task.process();
4          task.generateOutput();
5      } recover(ABoxException e) {
6          if(e.getCauseClass() == OutOfMemoryError.getClass()) {
7              // Upon OutOfMemoryError, propagate to terminate thread
8              throw e;
9          }
10     }
11 }

```

Fig. 4. Coordinated termination of a multi-threaded application upon an `OutOfMemoryError`. The named form of `abox` can be used to provide such recovery.

the content of the `execute` method in an `enter` block, which would specify that the code enters a failbox common to all threads.

We can also think about a variant of the above example that cannot be resolved using the failbox approach. Let us assume that, as the `task` object can configure itself before execution, it is also possible to reconfigure it to perform the same job using less memory but slower (e.g., by disabling an object pool). In such a case, the named form of the `abox` allows us to resolve the problem with the code in Figure 5 (again only by changing the content of the `execute` method). This solution is possible with the named form of `abox` since the `abox-recover` statement including the `abox` provides failure atomicity and coordinated exception handling. The failure atomicity of the property of the `abox-recover` statement allows the modifications of the execution inside the `abox` to be rolled back, thus the `task` object can be reverted to a consistent state where it can be reconfigured. The coordinated exception handling provided by the `abox-recover` statement allows the same behavior to be performed on all threads in a synchronized way and remedy the problem in a single step.

```

1  public void execute(Task task) {
2      abox("reconfigure", all) {
3          task.process();
4          task.generateOutput();
5      } recover(ABoxException e) {
6          if(e.getCauseClass() == OutOfMemoryError.getClass()) {
7              // Upon OutOfMemoryError, reconfigure and restart
8              task.reconfigure();
9          }
10     }
11 }

```

Fig. 5. Coordinated recovery to reconfigure tasks (for decreasing their memory footprint) upon `OutOfMemoryError`

4.2 Failure Mode Constructs

Since an atomic box corresponds to dependent code blocks, when an `abox` fails, its associated atomic box also fails. We call the atomic box that fails upon the failure of an `abox` an *active* atomic box. An active atomic box is defined as the set of `aboxes` of the same atomic box that have started executing and that have not yet started committing. This set is defined as long as at least one thread executes in the atomic box.

We argue that in terms of failure it is enough to consider an active atomic box rather than all the statically defined atomic box to ensure dependency-safety and failure atomicity. Since `aboxes` that have started committing are guaranteed not to execute on any inconsistent state that can be generated by the `aboxes` of the active atomic box (`aboxes` execute in isolation), their exclusion does not harm dependency-safety. Moreover, the consistency of data is ensured as long as the commit of `aboxes` that have started committing are allowed to finish before the `aboxes` of the active atomic box start performing recovery actions. So the rollback of an active atomic box does not require `aboxes` that have already started committing to rollback. Hence, it is safe to provide failure atomicity only for an active atomic box.

To have better understanding of the concept of active atomic box consider the solution proposed in Figure 4. For this solution if we think that the tasks executed by all of the threads have more or less the same load, the threads will generally be executing the `execute` method at about the same time periods. However, if we think of a scenario where tasks have variable load, this may not be true. So when the `OutOfMemoryError` is raised, some threads may be executing in the content of the `abox`, while some others may be still committing the `abox` in the `execute` method and some others maybe fetching a new task from the `taskQueue` (these threads have not yet entered in an `abox`). In such a case, the proposed solution may not stop all the threads since not all may be executing in the active atomic box when the `OutOfMemoryError` is raised. However, for these non-terminated threads the execution continues safely; threads that were committing while the exception is raised in active atomic box do not have any more dependence on the `aboxes` of the atomic box, and threads that have not yet entered execution in the atomic box may not raise an `OutOfMemoryError` if there is enough memory once the threads of the active atomic box get killed. Even if an `OutOfMemoryError` is again raised, this will be resolved by the active atomic box defined at the time of the second exception. Hence, we see that by applying the failure atomicity and dependency-safety only on the active atomic box it is also possible to provide safe executions.

The failure of an active atomic box results in the following coordinated behavior in the `aboxes` that constitute the active atomic box:

1. The `aboxes` that constitute an active atomic box switch to failure mode. This triggers the coordinated failure behavior of the atomic box.
2. All the `aboxes` that switch to failure mode automatically rollback. At the same time all `aboxes` that have started committing terminate their commit.

3. All the threads executing in an active atomic box are notified of a special exception `ABoxException` (the structure of this exception is explained later).
4. All the threads executing in an active atomic box wait for each other to make sure that they all rolled back and received the `ABoxException` notification. The threads in the active atomic box also wait for threads running an `abox` that have already started committing to finish their commit operation (which may not succeed and trigger an abort).
5. All the `aboxes` that constitute an active atomic box perform the recovery actions in the associated `recover` blocks according to the `ABoxException` they receive. Entry in the atomic box is forbidden for any thread during recovery.
6. All the threads executing in an active atomic box wait for each other to terminate their recovery actions. Once all recovery actions are terminated each of the threads executing in the active atomic box decide locally how to redirect their control after failure.

The `ABoxException`. The structure of the `ABoxException` that is notified to all the threads in the active atomic box is as follows:

```
public class ABoxException {
    Class causeClass;
    String message;
    Thread source;
    String aboxName;
    int handlingContext;
    // Methods omitted...
}
```

where the `causeClass` field stores the class of the exception raised by the `abox` that initially failed (initiator `abox`), the `message` field is the message of the original exception, the `source` field is the reference to the `Thread` object executing the initiator `abox`, `aboxName` is the name of the failing atomic box and `handlingContext` is an integer value that defines which of the corresponding `recover` blocks associated to the atomic box will be executed. The value of the `handlingContext` corresponds to the `<handlingContext>` parameter of the initiator `abox` (the details for the values of `handlingContext` are explained below together with the `recover` block). Note that the `ABoxException` stores the class of the original exception object that initiated the atomic box failure rather than its reference. This is a deliberate choice since the original exception object can include references to other objects that are allocated inside the initiator `abox` and that will be invalidated by the rollback performed upon the failure of the atomic box.

The `recover block`. A `recover` block encloses recovery actions to be executed when the `abox` it is associated to fails. Since the `recover` block is related to failure of an atomic box, it is only part of failure mode execution. Note also that the `recover` block does not execute in a transactional context; it always

executes after its corresponding `abox` rolls back. The decision of whether the `recover` block will be executed depends on the `handlingContext` parameter of `ABoxException` sent by the initiator `abox`. Two values exist for the parameter `handlingContext`: `local` and `all`. With the `local` option, only the `recover` block of the initiator `abox` will be executed, other threads will not execute any recovery action. If the `all` option is chosen all the threads executing in the atomic box execute their respective `recover` blocks.

Whichever of the `handlingContext` options is chosen, once the `recover` block executions are terminated each of the threads executing in the atomic box take their own control flow decision. If the `handlingContext` parameter has the value `local`, the initiator `abox` redirects the control flow according the control flow keyword used in its `recover` block (for the control flow keywords see Section 4.3). All the other threads in the atomic box re-execute the `abox` for which they perform recovery actions. If the `handlingContext` parameter has the value `all`, each of the threads redirects the control flow according the control flow keyword used in its respective `recover` block.

If the `recover` block of `abox-recover` statement has been omitted, the thread executing this `abox-recover` statement performs no recovery and re-executes the `abox` of the `abox-recover` statement.

The syntax of the `recover` block can be described as follows:

```
recover(ABoxException exceptionName) { S }
```

where the `exceptionName` is the name of the `ABoxException` notified to all the threads upon failure of an atomic box. The exception parameter of the `recover` block is expected to be of type `ABoxException` and providing an exception of another type will produce a compiler error.

Having analyzed most of the properties of the normal and failure modes, it would be appropriate to analyze the mechanisms described above in an example. At this point, we can use another variant of the running example of Figure 2 with an `OutOfMemoryError` being raised during the execution of the `execute` method. Suppose, in this case, that the programmer knows that he is using too many threads and if the heap allocated by the JVM is not enough, it would be enough for him to kill only some of the worker threads. This would effectively handle the exception while keeping the parallelism of thread executions at a reasonable level. Since the programmer would not know the size of the memory allocated in advance he can choose to implement the solution in Figure 6 using the atomic boxes.

The solution shown in Figure 6 is the same as the code in Figure 4 except that the name of the `<handlingContext>` parameter is set to `local` instead of `all`. With this change each time an `OutOfMemoryError` is raised only the thread raising the exception executes the `throw` statement and kills itself. This solution works better than a simple `try-catch` because with the `try-catch` solution multiple threads could have raised the same exception at the same time and, being unaware of the exceptions raised in other threads, all of these threads would kill themselves leaving a smaller amount of threads running in the system, rather than gradually decreasing the amount of concurrency. Gradual decrease is

```

1  public void execute(Task task) {
2      abox("killSome", local) {
3          task.process();
4          task.generateOutput();
5      } recover(ABoxException e) {
6          if(e.getCauseClass() == OutOfMemoryError.getClass()) {
7              // Upon OutOfMemoryError, propagate to terminate local thread
8              throw e;
9          }
10     }
11 }

```

Fig. 6. Coordinated recovery to decrease the memory used by the multi-threaded application by killing only some of the threads upon `OutOfMemoryError`

possible thanks to the coordinated nature of the exception handling: coordination imposes the threads to abort their `aboxes` (instead of killing themselves) and restart execution after the initiator `abox`'s thread is killed. Thanks to the failure atomicity provided by atomic boxes, this can safely be repeated as many times as required until the required number of threads are killed.

4.3 Redirecting Control Flow after Recovery

For providing control flow specific to `abox-recover` statement, we introduce two new control flow keywords: `leave` and `retry`. These keywords are to be used mainly inside `recover` blocks but they can also be used with similar semantics in the `aboxes`. The only difference of using the keywords in an `abox` is that they immediately fail the `abox` (and respectively also the active atomic box) and they behave as a `recover` block that has no other recovery actions but only the specified keyword. Thus, the existence of these keywords in the `abox` will just serve as a shortcut to a case where the atomic block has failed and we execute only a `leave` or `retry` inside the `recover` block.

If no control flow keyword is provided, upon exit, the `recover` block implicitly re-execute the associated active `abox`. A programmer can also explicitly ask for re-execution of the associated `abox` using the `retry` keyword. In contrast, a `leave` keyword will pass the control to the statement following the `recover` block. Note that with a `leave` keyword, the effect of an `abox` is as if it had never executed. The reason is that the failure of the `abox` has caused the rollback of the modifications performed within.

The use of `throw` statement inside `recover` block will quit the `recover` block and propagate the exception in the context of the statement following the `recover` block. With a `throw` statement, again the atomic box appears as if it has never executed. Similarly if an unhandled exception is raised in recovery action code enclosed in a `recover` block, the behavior is the same as an explicit `throw` statement.

Any already existing control flow keyword (except the `throw` keyword) that quits a block (i.e., `continue`, `break` and `return`) does not change semantics

with our language extension. When used inside an `abox` (and not used inside a nested block such as a loop) they imply immediate commit of the tentative modifications up to the point of occurrence of the keyword and pass the control to the target destination outside the `abox` and `recover` block. If those control keywords are used inside a `recover` block, they behave exactly the same way as in the `abox` except that, since the `abox` is rolled back, none of the effects of the `abox` are visible (but of course the modifications inside the `recover` block are effective).

The use of a `throw` statement inside the `abox` raises an exception in the block as in plain Java. If the exception is handled inside the `abox` the behavior of the `throw` statement is unchanged. However, if the exception is not handled in the `abox`, the `abox` (and the corresponding active atomic box) switches to failure mode.

4.4 Nesting of Atomic Boxes

The failure of an `abox` can also trigger the failure of an atomic box other than the one it belongs to. If the failing atomic box is parent of another atomic box, when the parent atomic box fails, the child atomic box also fails, thus both the parent and the child atomic boxes switch to failure mode. In contrast, when a child atomic box fails, its parent atomic box does not fail, thus the child atomic box switches to failure mode, while the parent atomic box does not.

The fact that atomic boxes have ascendants or descendants is reflected by a hierarchical naming of `aboxes`. The `name` parameter of an `abox` can be a string of the form `x.y.z` following the naming convention of Java package names.

4.5 Resolution of Concurrently Raised Exceptions

Up to this point we have considered only the case where a single `abox` initiates an atomic box failure. If an exception needs to be treated by an `abox`, this is most probably because the exception concerns all the threads executing in the atomic box. So it is not surprising to expect that multiple `aboxes` raise the same exception and fail the atomic box. It is also perfectly possible that different `aboxes` of the same atomic box, concurrently raise the different exceptions and cause the atomic box to fail.

The atomic box takes a very simple approach to resolve concurrently raised exceptions thanks to its failure atomicity property: an atomic box allows only one exception (the first one to be caught) to be treated in failure mode and ignores all the concurrently raised exceptions during failure mode.

The atomic box does not consider all the concurrently raised exceptions together. By handling one exception and removing its cause before re-execution, one may avoid other concurrent exceptions to occur again. During re-execution, if the cause of the concurrently raised exceptions are not removed they will again manifest and fail the atomic box. They will thus be treated during re-execution.

As can be noticed, among other advantages, the atomic box approach brings an elegant solution to the concurrent exception handling problem thanks to its failure atomicity property. Actually, the solution presented in Figure 6 is a

good example illustrating the resolution of concurrently raised exceptions. In this example, other than the coordinated nature of the exception handling, it is the simple concurrent exception handling approach taken by atomic boxes that allows us to kill only as many threads as required.

5 Atomic Boxes Implementation

We have implemented a concurrent exception handling compiler framework, called CXH, that supports the language constructs proposed in Section 4. The CXH compiler framework produces bytecode that is executable by any Java virtual machine in a three-step process. First it runs our pre-compiler, TMJAVA that converts the extended language into annotated Java code. The annotations are used to detect, in the bytecode, which parts of the code have the `abox` semantics. Second our CXH embeds the LSA transactional memory library [24] that provides wrappers to shared memory accesses. Our `aboxes` benefit from the speculative execution of TMs to ensure that no exceptions are raised before applying any change in the shared memory. Third, CXH uses an existing bytecode instrumentation framework, Deuce [33], that redirects calls within annotated methods to transactional wrappers. We describe below these three components in further detail.

5.1 Language Support for Atomic Boxes

We implemented TMJAVA, a Java pre-compiler that converts `abox-recover` constructs in annotated Java code. This allows us to compile the resulting code using any Java compiler. TMJAVA converts each `abox` into a dedicated method that is annotated with an `@Atomic` keyword. More precisely, TMJAVA analyzes the code to find the `aboxes` (`abox` keyword) inside class methods. Then, for each such `abox` it creates a new method whose body is the content of the corresponding `abox` and replaces the original `abox` with a call to this new method. The conversion of an `abox` a into a method m requires passing some variables to the produced method m to address the following issues:

1. Variables that belong to the context of the method enclosing the `abox` a should also be accessible inside the scope of the produced method m .
2. Variables that belong to the context of the method enclosing the `abox` a and that are modified inside a should have their modifications effective outside the produced method m (as it would be for `abox` a).

To ensure that variables are still visible inside the produced methods, the variables whose scope are out of `abox` context are passed as input parameters to the corresponding method. For the state of variables to be reflected outside the scope of the `abox`, these variables are passed as parameters using arrays (if the variables are of primitive types). When the method returns, we copy back these array elements into the corresponding variables.

The resulting annotated Java code can be compiled using any Java compiler. TMJAVA is available for download from <http://tmware.org/tmjvba>.

5.2 Transactional Memory Wrappers

We use LSA [24], an efficient time-based transactional memory algorithm that maps each shared memory location with a timestamp. Each transaction of LSA executes speculatively by buffering its modifications. If a transactions reaches its end without having aborted, it attempts to commit by applying its modifications to shared memory. More precisely, when a transaction starts it records the value of a global time base, implemented as a shared counter. Upon writing a shared location, the transaction acquires an associated ownership record, buffers the write into a log, and continues executing subsequent accesses. At the end, when the transaction tries to commit, it reports all the logged writes in memory by writing the value, incrementing the global counter, and associating its new version to all written locations as part of the ownership records. Upon reading a shared location, it first checks if the location is locked (and aborts if locked), then compares the version of the location to the counter value it has seen. If the location has a higher version than this value, this means that a concurrent transaction has modified the location, indicating a conflict.

The particularity of the LSA algorithm is to allow the transaction to commit despite such a conflict thanks to incremental validation: if all previously read values are still consistent, i.e., their versions have not changed since they have been read, the transaction has a valid consistent snapshot and can resume without aborting.

Our `abox` leverages memory transactions that execute speculatively on shared data. The main difference between `aboxes` and the transactions lies in the fact that each `abox` decides whether to abort or commit its changes also depending on (concurrent) exceptions raised. Before committing, an `abox` makes sure that no exception was raised inside the block or by a dependent `abox`.

5.3 Bytecode Instrumentation

After compilation we obtain a bytecode where annotated methods directly access the memory. To ensure that these annotated methods, which correspond to the original `aboxes`, execute speculatively we have to redirect their memory accesses to the transactional memory. To that end, we use the Deuce framework [33] to instrument the annotated method calls at load time. Deuce instruments class methods annotated with `@Atomic` such that accesses to shared data inside those methods are performed transactionally. This bytecode instrumentation redirects all `abox` memory accesses to LSA so that each `abox` executes as a transaction.

6 Evaluation

We compare our `abox` solution against failbox [3] on an Intel Core2 CPU running at 2.13GHz. It has 8-way associative L1 caches of 32KB and an 8-way associative L2 cache of 2MB. For `abox` we implemented the compiler framework as explained in Section 5 whereas for failboxes we reused the original code from [3].

6.1 Producer-Consumer Example

Our first experiments consist of a simple producer-consumer application, where one thread pushes an item to a shared stack while another pops the topmost item from the same stack. For the sake of evaluation, the stack `push()` method raises an exception if adding the new item to the stack would exceed its capacity. We evaluated two versions of the same program: one using `failbox`, the other using our `abox`. The execution time of these two versions has been evaluated in normal cases (where we fill the stack prior to execution such that no exceptions are raised) and for handling exceptions (where we try to push an item to an already full stack). Results are averaged over 100 executions.

Table 1. Execution times of `abox` and `failbox` (in microseconds) on a multi-threaded producer-consumer application when no exceptions are raised

	min	max	average
<code>abox</code>	7.27	11.67	8.92
<code>failbox</code>	15.70	34.97	18.58
speedup of <code>abox</code>	1.34	4.81	2.08

Tables 1 and 2 report the minimum, maximum and average execution time in microseconds, respectively without and with exceptions. On the one hand, we observe that our solution executes about $2\times$ faster (on average) than `failboxes` in normal executions. This is due to a cache effect observed with `failbox` approach. Each time a `failbox` is entered a shared variable is checked to verify whether it has failed. Since this experiment requires very frequent entries to a `failbox` by multiple threads the `failbox` entries are serialized. Our implementation does not suffer from this problem since the check for the failure of an `abox` does not need to be verified often (an `abox` is executed in isolation from other code).

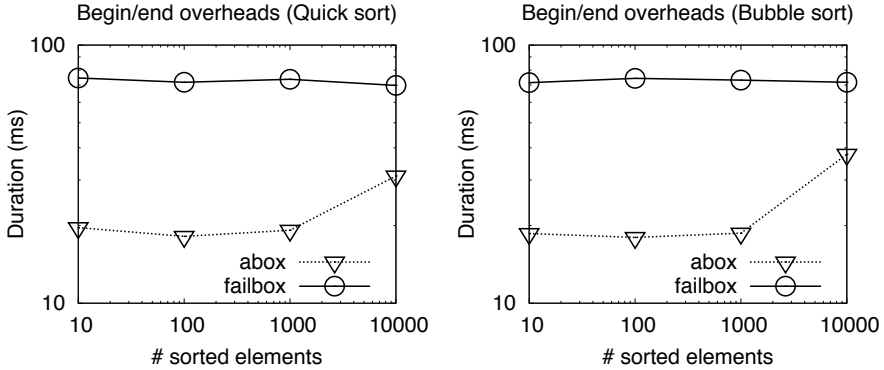
On the other hand, our solution performs more than $15\times$ faster (on average) than `failboxes` to handle exceptions. We conjecture that it is due to the fact that `failbox` approach uses the `interrupt` mechanism to communicate the exception on one thread to the other threads. The `abox` approach communicates over the shared memory, resulting in a faster notification. It is worth mentioning that our `aboxes` permit both `push()` and `pop()` methods to recover from exception, allowing the program to resume, while `failbox` simply stops the program upon the first exception raised. Considering this desirable behavior and the observed overhead, `abox` clearly represents a promising approach.

6.2 Sorting Examples

Our second experiments rely on two single-threaded sorting applications (quick-sort and bubble-sort) coded in 3 ways: (i) using plain Java (with no extensions), (ii) inside `failboxes`, and (iii) inside `abox` blocks. The plain Java version is used

Table 2. Execution times of `abox` and `failbox` (in microseconds) on a multi-threaded producer-consumer application when exceptions are raised

	min	max	average
<code>abox</code>	1.40	2.62	2.22
<code>failbox</code>	32.167	47.23	34.55
speedup of <code>abox</code>	12.28	33.74	15.7

**Fig. 7.** Comparison of the overhead produced when starting and terminating an `abox` and a `failbox` (note the logarithmic scales on both axes)

to measure the inherent overhead of `failbox` and `abox` versions. The sort is performed inside a function and the application can choose to run either a *quick-sort* or a *bubble-sort* function.

Figures 7 through 9 depict the performance of `failbox` and `abox` on quick-sort (left column) and bubble-sort (right column). Figure 7 compares the execution overhead due to entering and leaving an `abox` block or a `failbox` (we call this *begin/end overhead*). Figure 8 shows the execution time performance of `abox` and `failbox` executions without the *begin/end overhead*. Figure 9 depicts the total execution time performance of `abox` and `failbox`. The execution time performance depicted in figures 8 and 9 are given as the slowdown with respect to the performance of the plain Java version, which does not have any begin/end overhead. Each point in the graphs corresponds to the average of 10 runs.

The results show that although the `failbox` approach performs as good as plain Java inside the `failbox`, its begin/end overhead is quite high. We attribute this high overhead of the `failbox` approach to the memory allocation performed to generate a new `failbox` (be it a child or a new `failbox`) before entering the `failbox`. Figure 9 also illustrates that `abox` blocks perform better than the `failbox` approach for input arrays of up to about 1000 elements. This demonstrates that our `abox` implementation, although using transactions to sort array elements, performs well even compared to simpler approaches that do not roll back state changes.

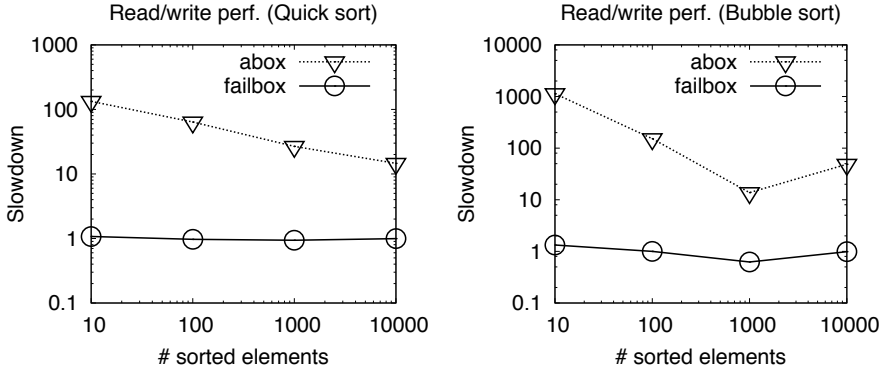


Fig. 8. Comparison of the overhead due to accessing the shared memory in `abox` and `failbox` (note the logarithmic scales on both axes)

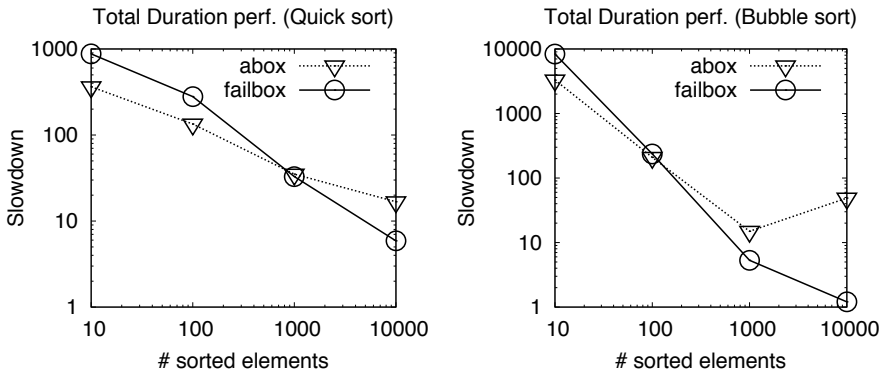


Fig. 9. Comparison of the total duration time of `abox` and `failbox` (note the logarithmic scales on both axes)

7 Conclusion and Future Work

This paper introduces language constructs for concurrent exception handling, a way to handle exceptions in a concurrent manner for multi-threaded software.

The key novelty is to ensure that any inconsistent state resulting from an exception cannot be accessed by concurrent threads, thus allowing the programmer to define concurrent exception handlers. The alternative `failbox` [3] language construct that prevents threads from running on inconsistent states simply stops all threads. Letting the programmer define concurrent exception handlers allows us to recover rather than stop. For example, the programmer can remedy the cause of an exception and retry the concurrent execution.

To experiment with our solution, we have implemented a compiler framework, CXH, for our language constructs that converts `aboxes` into code that uses an underlying software transactional memory runtime. Our preliminary evaluations

indicate that the overhead of our transactional wrappers is low: when accessing up to hundreds of elements, `aboxes` execute twice faster than `failboxes`.

The fact that the transactional memory overhead does not significantly impact the concurrent exception handling should encourage further research in this direction. This work could for example benefit from ongoing progress in hardware and hybrid transactional memory to further reduce overheads, as our current implementation is purely software based. Even though there is a long road before integrating such language constructs in Java, we believe that exploring transactional memory as a building block for concurrent exception handling will raise new interesting research challenges and offer new possibilities for programmers.

Acknowledgements. This work is supported in part by the Swiss National Foundation under grant 200021-118043 and the European Union's Seventh Framework Programme (FP7/2007-2013) under grants 216852 and 248465.

References

1. Cabral, B., Marques, P.: Exception handling: A field study in Java and.NET. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 151–175. Springer, Heidelberg (2007)
2. Stelting, S.: Robust Java: Exception Handling, Testing and Debugging. Prentice Hall, New Jersey (2005)
3. Jacobs, B., Piessens, F.: Failboxes: Provably safe exception handling. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 470–494. Springer, Heidelberg (2009)
4. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edn. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Francisco (2010)
5. Spector, A.Z., Daniels, D.S., Duchamp, D., Eppinger, J.L., Pausch, R.F.: Distributed transactions for reliable systems. In: SOSP, pp. 127–146 (1985)
6. Liskov, B.: Distributed programming in Argus. *Commun. ACM* 31, 300–312 (1988)
7. Haines, N., Kindred, D., Morrisett, J.G., Nettles, S.M., Wing, J.M.: Composing first-class transactions. *ACM Trans. Program. Lang. Syst.* 16, 1719–1736 (1994)
8. Jimenez-Peris, R., Patino-Martinez, M., Arevalo, S., Peris, R.J., Ballesteros, F., Carlos, J.: Translib: An Ada 95 object oriented framework for building transactional applications (2000)
9. Kienzle, J., Romanovsky, A.: Combining tasking and transactions, part II: Open multithreaded transactions. In: IRTAW 2000, pp. 67–74 (2001)
10. Xu, J., Randell, B., Romanovsky, A., Rubira, C.M.F., Stroud, R.J., Wu, Z.: Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: FTCS, pp. 499–508 (1995)
11. Capozucca, A., Guelfi, N., Pelliccione, P., Romanovsky, A., Zorzo, A.F.: Frameworks for designing and implementing dependable systems using coordinated atomic actions: A comparative study. *J. Syst. Softw.* 82, 207–228 (2009)
12. Beder, D.M., Randell, B., Romanovsky, A., Rubira, C.M.F.: On applying coordinated atomic actions and dependable software architectures for developing complex systems. In: ISORC, pp. 103–112 (2001)

13. Zorzo, A.F., Stroud, R.J.: A distributed object-oriented framework for dependable multiparty interactions. In: OOPSLA, pp. 435–446 (1999)
14. Filho, F., Rubira, C.M.F.: Implementing coordinated error recovery for distributed object-oriented systems with AspectJ. *J. of Universal Computer Science* 10(7), 843–858 (2004)
15. Ziarek, L., Schatz, P., Jagannathan, S.: Stabilizers: A modular checkpointing abstraction for concurrent functional programs. In: ICFP, pp. 136–147 (2006)
16. Scherer III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC, pp. 240–248 (2005)
17. Guerraoui, R., Herlihy, M., Kapalka, M., Pochon, B.: Robust contention management in software transactional memory. In: SCOOL (2005)
18. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA, pp. 289–300 (1993)
19. Shavit, N., Touitou, D.: Software transactional memory. In: PODC, pp. 204–213 (1995)
20. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC, pp. 92–101 (2003)
21. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA, pp. 388–402 (2003)
22. Dalessandro, L., Marathe, V., Spear, M., Scott, M.: Capabilities and limitations of library-based software transactional memory in C++. In: Transact (2007)
23. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
24. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 284–298. Springer, Heidelberg (2006)
25. Felber, P., Gramoli, V., Guerraoui, R.: Elastic transactions. In: Keidar, I. (ed.) DISC 2009. LNCS, vol. 5805, pp. 93–107. Springer, Heidelberg (2009)
26. Shinnar, A., Tarditi, D., Plesko, M., Steensgaard, B.: Integrating support for undo with exception handling. Technical Report MSR-TR-2004-140, Microsoft Research (2004)
27. Cabral, B., Marques, P.: Implementing retry - featuring AOP. In: Fourth Latin-American Symposium on Dependable Computing, pp. 73–80 (2009)
28. Harris, T.: Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.* 58(3), 325–343 (2005)
29. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: PPOPP, pp. 48–60 (2005)
30. Fetzer, C., Felber, P.: Improving program correctness with atomic exception handling. *J. of Universal Computer Science* 13(8), 1047–1072 (2007)
31. Volos, H., Tack, A.J., Goyal, N., Swift, M.M., Welc, A.: xCalls: Safe I/O in memory transactions. In: EuroSys, pp. 247–260 (2009)
32. Porter, D.E., Hofmann, O.S., Rossbach, C.J., Benn, A., Witchel, E.: Operating system transactions. In: SOSP, pp. 161–176 (2009)
33. Korland, G., Shavit, N., Felber, P.: Deuce: Noninvasive software transactional memory in Java. *Transactions on HiPEAC* 5(2) (2010)

Author Index

- Adve, Vikram S. 306
Aldrich, Jonathan 2, 459
- Balakrishnan, Gogul 583
Baltopoulos, Ioannis G. 484
Balzer, Stephanie 358
Beckman, Nels E. 2
Bergel, Alexandre 533
Bhattacharya, Suparna 408
Bocchino Jr., Robert L. 306
Borgström, Johannes 484
Burg, Brian 52
Butler, Simon 130
- Carbin, Michael 609
Chambers, Craig 229
Chen, Yuting 510
Chis, Adriana E. 383
- Dietl, Werner 333
Duala-Ekoko, Ekwa 79
- Ernst, Michael D. 333
- Felber, Pascal 634
- Garcia, Ronald 459
Giarrusso, Paolo G. 155
Gopinath, K. 408
Gordon, Andrew D. 484
Gramoli, Vincent 634
Gross, Thomas R. 358
Gupta, Aarti 583
Gupta, Manish 408
- Hammer, Christian 52
Harmanci, Derin 634
Hauswirth, Matthias 27
- Igarashi, Atsushi 433
Ivančić, Franjo 583
- Kaehler, Ted 179
Kästner, Christian 155
Kay, Alan 179
- Kim, Duri 2
Kling, Michael 609
Kollee, Christian 255
- Maeda, Naoto 583
Maoz, Shahar 230, 281
Meijer, Erik 1
Mileva, Yana Momchilova 105
Misailovic, Sasa 609
Mitchell, Nick 383
Müller, Peter 333
Murphy, John 383
- Nanda, Mangala Gowri 408
- Ohshima, Yoshiki 179
Ostermann, Klaus 155
O'Sullivan, Patrick 383
- Parsons, Trevor 383
Pilgrim, Jens von 255
Pothier, Guillaume 553
Prabhu, Prakash 583
- Rendel, Tillmann 155
Richards, Gregor 52
Rinard, Martin C. 609
Ringert, Jan Oliver 230, 281
Robillard, Martin P. 79
Rumpe, Bernhard 230, 281
- Schonberg, Edith 383
Schwartzbach, Michael I. 434
Sevitsky, Gary 383
Sharp, Helen 130
Steimann, Friedrich 255
- Tanter, Éric 459, 558
Taube-Schock, Craig 204
- Vitek, Jan 52
- Walker, Robert J. 204
Warth, Alessandro 179
Wasykowski, Andrzej 105

Wermelinger, Michel 130
Winther, Johnni 434
Witten, Ian H. 204
Wolff, Roger 459

Xu, Hao 510

Yu, Yijun 130

Zaparanuks, Dmitrijs 27
Zeller, Andreas 105
Zhang, Cheng 510
Zhang, Sai 510
Zhao, Jianjun 510