Journal Subline

LNCS 8400

Eric Bodden · Shahar Maoz
Jörg Kienzle
Guest Editors

Transactions on Aspect-Oriented Software Development XI

Shigeru Chiba · Éric Tanter
Editors-in-Chief

Springer

Lecture Notes in Computer Science 8400

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

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

Shigeru Chiba Éric Tanter
Eric Bodden Shahar Maoz Jörg Kienzle (Eds.)

Transactions on Aspect-Oriented Software Development XI

Springer

Editors-in-Chief

Shigeru Chiba
The University of Tokyo, Tokyo, Japan
E-mail: chiba@acm.org

Éric Tanter
University of Chile, Santiago, Chile
E-mail: etanter@dcc.uchile.cl

Guest Editors

Eric Bodden
Technical University of Darmstadt, Darmstadt, Germany
E-mail: eric.bodden@ec-spride.de

Shahar Maoz
Tel Aviv University, Tel Aviv, Israel
E-mail: maoz@cs.tau.ac.il

Jörg Kienzle
McGill University, Montreal, Canada
E-mail: joerg.kienzle@mcgill.ca

ISSN 0302-9743 (LNCS)
e-ISSN 1611-3349 (LNCS)
ISSN 1864-3027 (TAOSD)
e-ISSN 1864-3035 (TAOSD)
ISBN 978-3-642-55098-0
e-ISBN 978-3-642-55099-7
DOI 10.1007/978-3-642-55099-7
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2014936185

© Springer-Verlag Berlin Heidelberg 2014
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in ist current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, 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.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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)

Editorial

Welcome to Volume XI of the Transactions on Aspect-Oriented Software Development. This volume has two special sections on Runtime Verification and Analysis and on the Best Papers of AOSD 2013.

The first section, guest edited by Eric Bodden and Shahar Maoz, has two excellent papers. It highlights runtime verification as a *killer* application of aspect-orientation. The second section has five papers and is guest edited by Jörg Kienzle, who was Program Committee Chair of the Modularity:aosd 2013 conference held in Fukuoka, Japan. Modularity:aosd 2013 (AOSD 2013 in short) constituted a premier forum for researchers and practitioners to present their work and discuss technical challenges on advanced software modularity and, in particular, aspect-oriented software development. Although the conference proceedings is already a collection of high-quality papers in this area, which are available from ACM digital library, this special section collects longer versions of the best papers presented at the conference.

We thank all the guest editors for soliciting submissions, running review processes, and collecting final versions within such a short period. We are pleased to publish this special issue in a timely fashion. We also thank the editorial board members for their continued guidance and input on the policies of the journal, the reviewers for volunteering a significant amount of time despite their busy schedules, and the authors who submitted papers to the journal.

February 2014

Shigeru Chiba
Éric Tanter
Editors-in-Chief

Guest Editors' Foreword

Special Section on Runtime Verification and Analysis

This special section of TAOSD assembles novel contributions that link together the fields of Aspect-Oriented Software Development and Runtime Verification and Analysis. For more than a decade, researchers and practitioners have wondered about a so-called "killer application" for aspect-oriented programming (AOP). Runtime verification (RV) has turned out to be one of the most convincing use cases for AOP. Runtime verification tools typically instrument a program under test with an oracle, defined through a high-level specification language such as a temporal logic or model-based formalism. Prior to the advent of AOP, all of those tools resorted to manual program instrumentation, on source code, bytecode or machine code. This made tools and approaches hard to understand, compare, compose and prove correct. Since then, the advent of AOP has radically changed the way in which tools for RV and dynamic analysis are developed. Nowadays, many tools instrument programs by generating aspects from domain-specific textual descriptions or visual models, or use a programming model based on AOP concepts. But research results also flow in the other direction: requirements for RV and dynamic analysis tools have led to novel AOP language constructs improving the performance, modularity and maintainability of the analysis code. This special section presents two thoroughly peer-reviewed papers that give a high-quality example of the synergies that exist between these important fields of research.

February 2014

Éric Bodden
Shahar Maoz
Guest Editors

Guest Editor's Foreword

Special Section on the Best Papers of AOSD 2013

This special section of TAOSD gathers revised and extended versions of the best papers presented at Modularity:aosd 2013, the 12th International Conference on Modularity and Aspect-Oriented Software Development. The papers were selected based on their evaluation by the Modularity:aosd 2013 Program Committee, of which I served as Program Committee Chair. The authors of the top five papers were invited to submit a revised and extended version of their work. Each revised paper was evaluated both by a member of the Modularity:aosd 2013 PC and by an external reviewer, in order to ensure that the extensions were relevant, consistent, substantial and well-integrated. Four articles were accepted after only one round requesting a minor revision, one paper went through two rounds of revision. Since two of the papers are co-authored by editors-in-chief of the journal, all the reviewing was handled exclusively by myself without involvement of the editors-in-chief.

As a result of the constructive effort of both authors and reviewers, this issue is a high-quality snapshot of current state-of-the-art research related to modularity, covering topics as varied as formal methods and type systems, static analysis approaches for software architectures, model-driven engineering and model composition, as well as aspect-oriented programming, event-driven programming, and reactive programming.

February 2014

Jörg Kienzle
Guest Editor

Editorial Board

Mehmet Akşit	University of Twente, The Netherlands
Shigeru Chiba	The University of Tokyo, Japan
Siobhán Clarke	Trinity College Dublin, Ireland
Robert Filman	Google, USA
Wouter Joosen	Katholieke Universiteit Leuven, Belgium
Shmuel Katz	Technion-Israel Institute of Technology, Israel
Gregor Kiczales	University of British Columbia, Canada
Gary T. Leavens	University of Central Florida, USA
Karl Lieberherr	Northeastern University, USA
Mira Mezini	Darmstadt University of Technology, Germany
Ana Moreira	New University of Lisbon, Portugal
Harold Ossher	IBM Research, USA
Klaus Ostermann	University of Marburg, Germany
Awais Rashid	Lancaster University, UK
Douglas C. Schmidt	Vanderbilt University, USA
Mario Südholt	Ecole des Mines de Nantes, France
Éric Tanter	University of Chile, Chile

Table of Contents

Runtime Verification and Analysis

Run-Time Assertion Checking of Data- and Protocol-Oriented Properties of Java Programs: An Industrial Case Study ... 1
Frank S. de Boer, Stijn de Gouw, Einar Broch Johnsen, Andreas Kohn, and Peter Y.H. Wong

Event Modules: Modularizing Domain-Specific Crosscutting RV Concerns ... 27
Somayeh Malakuti and Mehmet Akşit

Best Papers of AOSD 2013

Method Slots: Supporting Methods, Events, and Advices by a Single Language Construct ... 70
YungYu Zhuang and Shigeru Chiba

Modularity and Dynamic Adaptation of Flexibly Secure Systems: Model-Driven Adaptive Delegation in Access Control Management ... 109
Phu H. Nguyen, Gregory Nain, Jacques Klein, Tejeddine Mouelhi, and Yves Le Traon

Effective Aspects: A Typed Monadic Embedding of Pointcuts and Advice ... 145
Ismael Figueroa, Nicolas Tabareau, and Éric Tanter

Modular Specification and Checking of Structural Dependencies ... 193
Ralf Mitschke, Michael Eichberg, Mira Mezini, Alessandro Garcia, and Isela Macia

Towards Reactive Programming for Object-Oriented Applications ... 227
Guido Salvaneschi and Mira Mezini

Author Index ... 263

Run-Time Assertion Checking of Data- and Protocol-Oriented Properties of Java Programs: An Industrial Case Study

Frank S. de Boer[1,2], Stijn de Gouw[1,2],
Einar Broch Johnsen[3], Andreas Kohn[4], and Peter Y.H. Wong[4]

[1] CWI, Amsterdam, The Netherlands
[2] Leiden University, The Netherlands
[3] University of Oslo, Norway
[4] Fredhopper B.V., Amsterdam, The Netherlands

Abstract. Run-time assertion checking is one of the useful techniques for detecting faults, and can be applied during any program execution context, including debugging, testing, and production. In general, however, it is limited to checking state-based properties. We introduce SAGA, a general framework that provides a smooth integration of the specification and the run-time checking of both data- and protocol-oriented properties of Java classes *and interfaces*. We evaluate SAGA, which combines several state-of-the art tools, by conducting an industrial case study from an eCommerce software company Fredhopper.

1 Introduction

Run-time assertion checking is one of the most useful techniques for detecting faults, and can be applied during any program execution context, including debugging, testing, and production [7]. Compared to program logics, run-time assertion checking emphasizes *executable specifications*. Assertions in general are clearly not executable in the sense that one cannot decide whether they hold (in the presence of unbounded quantification). As a result, for run-time assertion checking one has to restrict the class of assertions to executable ones. Whereas program logics are generally applied statically to cover all possible execution paths, which is in general undecidable, run-time assertion checking is a fully automated, on-demand validation process which applies to the actual runs of the program.

By their very nature, assertions are state-based in that they describe properties of the program variables, e.g. fields of classes and local variables of methods. In general, assertions as supported for example by the Java programming language or the Java Modeling Language (JML) [3] cannot be used to specify the *interaction protocol* between objects, in contrast to other formalisms such as message sequence charts and UML sequence diagrams. Consequently, existing state-of-the-art program logics for Java are not suited for proving protocol

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 1–26, 2014.
© Springer-Verlag Berlin Heidelberg 2014

properties. Moreover state-based assertions cannot be used to specify interfaces since interfaces do not have a state[1].

The main contribution of this paper is twofold. First, we introduce SAGA (Software trace Analysis using Grammars and Attributes), a run-time checker that provides a smooth integration of the specification and the run-time checking of both data- and protocol-oriented properties of Java classes *and interfaces.* SAGA combines four different components: a state-based assertion checker, a monitoring tool, a meta-programming tool, and a parser generator. Aspect-oriented programming is tailored for monitoring, and in contrast to transformations source of Java code or debugger-based solutions [9] it is designed for high performance applications and supports the monitoring of precompiled libraries for which no source code is available. The tool can be used for run-time checking of any Java program, which requires specific support for the main features listed in Table 1, as discussed in more detail in the following section. Secondly, we evaluate SAGA by conducting an industrial case study from the eCommerce software company Fredhopper.

Table 1. Supported features

Constructors
Inheritance
Dynamic Binding
Overloading
Static Methods
Access Modifiers

The basic idea underlying SAGA is the representation of message sequences as words of a language generated by a grammar. Grammars allow, in a declarative and highly convenient manner, the description of the *protocol structure* of the communication events. However, the question is how to integrate such grammars with the run-time checking of assertions, and how to describe the *data flow* of a message sequence, i.e., the properties of the data communicated. We propose a formal modeling language for the specification of sequences of messages in terms of *attribute grammars* [14]. Attribute grammars allow the high-level specification of the data-flow of message sequences (e.g., their length) in terms of user-defined attributes of non-terminals. SAGA supports the run-time checking of assertions about these attributes (e.g., that the length of a sequence is bounded). This involves parsing the generated sequences of messages. These sequences themselves are recorded by means of a fully automated instrumentation of the given program by AspectJ[2].

[1] JML uses model variables for interface specifications. However, a separate represents clause is needed for a full specification, and such clauses can only be defined once an implementation has been given (and is not implementation independent).

[2] `www.eclipse.org/aspectj`

2 The Modeling Framework

Abstracting from implementation details (such as field values of objects), an execution of a Java program can be represented by its *global communication history*: the sequence of *messages* corresponding to the invocation and completion of (possibly static) methods. Similarly, the execution of a single object can be represented by its *local communication history*, which consists of all messages sent and received by that object. The behavior of a program (or object) can then be defined as the set of its allowed histories. Whether a history is allowed depends in general both on data (the contents of the messages, e.g. parameter and return values of method calls) and protocol (the order between messages). The question arises how such allowed sets of histories can be defined conveniently. In this section we show how attribute grammars provide a powerful and declarative way to define such sets. We use the interface of the Java `BufferedReader` (Figure 1) as a running example to explain the basic modeling concepts. In particular, we formalize the following property:

> The `BufferedReader` may only be closed by the same object which created it, and reads may only occur between the creation and closing of the `BufferedReader`.

```
interface BufferedReader {
  void close();
  void mark(int readAheadLimit);
  boolean markSupported();
  int read();
  int read(char[] cbuf, int off, int len);
  String readLine();
  boolean ready();
  void reset();
  long skip(long n);
}
```

Fig. 1. Methods of the BufferedReader Interface

As a naive first step one might be tempted to define the behavior of `BufferedReader` objects simply in terms of ‘call-m($\overline{T}$)’ and ‘return-m($\overline{T}$)’ messages of all methods ‘m’ in its interface, where the parameter types $\overline{T}$ are included to distinguish between overloaded methods (such as `read`). However, interfaces in Java contain only signatures of provided methods: methods where the `BufferedReader` is the callee. Calls to these methods correspond to messages received by the object. In general the behavior of objects also depends on messages sent by that object (i.e., where the object is the caller), and on the particular constructor (with parameter values) that created the object. Moreover,

it is often useful to select a particular subset of method calls or returns, instead of using calls and returns to all methods (a partial or incomplete specification). Finally, in referring to messages it is cumbersome to explicitly list the parameter types. A *communication view* addresses these issues.

2.1 Communication View

A communication view is a partial mapping which associates a name to each message. Partiality makes it possible to filter irrelevant events and message names are convenient in referring to messages.

Suppose we wish to specify that the `BufferedReader` may only be closed by the same object which created it, and that reads may only occur between the creation and closing of the `BufferedReader`. This is a property which must hold for the local history of all instances of `java.util.BufferedReader`. The communication view in Figure 2 selects the relevant messages and associates them with intuitive names: *open*, *read*, and *close*.

```
local view BReaderView grammar BReader.g
specifies java.util.BufferedReader {
  BufferedReader(Reader in) open,
  BufferedReader(Reader in, int sz) open,
  call void close() close,
  call int read() read,
  call int read(char[] cbuf, int off, int len) read
}
```

Fig. 2. Communication view of a BufferedReader

All return messages and call messages methods not listed in the view are filtered. Note how the view identifies two different messages (calls to the overloaded read methods) by giving them the same name *read*. Though the above communication view contains only provided methods (those listed in the `BufferedReader` interface), required methods (e.g., methods of other interfaces or classes) are also supported. Since such messages are sent to objects of a different class (or interface), one must include the appropriate type explicitly in the method signature. For example consider the following message:

```
call void C.m() out
```

If we would additionally include the above message in the communication view, all call-messages to the method `m` of class `C` sent by a `BufferedReader` would be selected and named *out*. In general, incoming messages received by an object correspond to calls of provided methods and returns of required methods. Outgoing messages sent by an object correspond to calls of required methods

and returns of provided methods. Incoming call-messages of local histories never involve static methods, as such methods do not have a callee.

Besides normal methods, communication views can contain signatures of constructors (i.e., the messages named *open* in our example view). Incoming calls to provided constructors raise an interesting question: what would happen if we select such a message in a local history? At the time of the call, the object has not even been created yet, so it is unclear which `BufferedReader` object receives the message. We therefore only allow return-messages of provided constructors (clearly required constructors do not pose the same problem, and consequently we allow selecting both calls and returns to required constructors), and for convenience omit **return**. Alternatively one could treat constructors like static methods, disallowing incoming call-messages to constructors in local histories altogether. However, this makes it impossible to express certain properties (including the desired property of the `BufferedReader`) and has no advantages over the approach we take.

Java programs can distinguish methods of the same name only if their parameter types are different. Communication views are more fine-grained: methods can be distinguished also based on their return type or their access modifiers (such as **public**). For instance, consider a scenario with suggestively named classes `Base` and three subclasses `Sub1`, `Sub2`, and `Sub3`, all of which provide a method `m`. The return type of `m` in the `Base`, `Sub1` and `Sub2` classes is the class itself (i.e., `Sub1` for `m` provided by `Sub1`). In the `Sub3` class the return type is `Sub1`. To monitor calls to `m` only with return type `Sub1`, simply include the following event in the view:

```
call Sub1 C.m() messagename
```

Local communication views, such as the one above, selects messages sent and received by *a single object* of a particular class, indicated by 'specifies java.util.BufferedReader'. In contrast, global communication views select messages sent and received by *any* object during the execution of the Java program. This is useful to specify global properties of a program. In addition to instance methods, calls and returns of static methods can also be selected in global views. Figure 3 shows a global view which selects all returns of the method `m` of the `Ping` class or interface or any of its subclasses, and all calls of the `Pong` class (or interface) or its subclasses. Note that communication views do not distinguish instances of the same class (e.g., calls to 'Ping' on two different objects of class 'Ping' both get mapped to the same terminal 'ping'). Different instances can be distinguished in the grammar using the built-in attributes 'caller' or 'callee'.

In contrast to interfaces of the programming language, communication views can contain constructors, required methods, static methods (in global views) and can distinguish methods based on return type or method modifiers such as 'static', or 'public'. See Table 1 for a list of supported features.

```
global view PingPong grammar pingpong.g {
  return void Ping.m() ping,
  call void Pong.m() pong
}
```

Fig. 3. Global communication view

2.2 Grammars

Context-free grammars provide a convenient way to define the protocol behavior of the allowed histories. The context-free grammar underlying the attribute grammar in Figure 4 generates the valid histories for `BufferedReader`, describing the prefix closure of sequences of the terminals 'open', 'read', and 'close' as given by the regular expression (open read* close). In general, the message names form the terminal symbols of the grammar, whereas the non-terminal symbols specify the structure of valid sequences of messages. In our approach, a communication history is valid if and only if it and all its prefixes are generated by the grammar.

For a justification of this approach, see the next discussion section. While context-free grammars provide a convenient way to specify the *protocol structure* of the valid histories, they do not take data such as parameters and return values of method calls and returns into account. Thus the question arises how to specify the *data-flow* of the valid histories. To that end, we extend the grammar with attributes. Each terminal symbol has *built-in* attributes named **caller**, **callee** and the parameter names for respectively the object identities of the caller, callee and actual parameters. Terminals corresponding to method returns additionally have an attribute **result** containing to the return value. In summary, the (built-in) attributes of terminals are determined from the method signatures. Non-terminals have *user-defined* attributes to define data properties of sequences of terminals. However, the attributes themselves do not alter the language generated by the attribute grammar, they only *define* properties of data-flow of the history. We extend the attribute grammar with assertions to specify properties of attributes. For example, in the attribute grammar in Figure 4 a user-defined synthesized attribute 'c' for the non-terminal 'C' is defined to store the identity of the object which closed the `BufferedReader` (and is **null** if the reader was not closed yet). Synthesized attributes define the attribute values of the non-terminals on the left-hand side of each grammar production, thus the 'c' attribute is not set in the productions of the start symbol 'S'.

The assertion allows only those histories in which the object that opened (created) the reader is also the object that closed it. Throughout the paper the start symbol in any grammar is named 'S'. For clarity, attribute definitions are written between parentheses '(' and ')' whereas assertions over these attributes are surrounded by braces '{' and '}'.

```
S ::= open C1 {assert (open.caller == null || open.caller == C1.c ||
                        C1.c == null);}
    | ε
C ::= read C1  (C.c = C1.c;)
    | close S  (C.c = close.caller;)
    | ε        (C.c = null;)
```

Fig. 4. Attribute Grammar which specifies that 'read' may only be called in between 'open' and 'close', and the reader may only be closed by the object which opened it

Assertions can be placed at any position in a production rule and are evaluated at the position they were written. Note that assertions appearing directly before a terminal can be seen as a precondition of the terminal, whereas post-conditions are placed directly after the terminal. This is in fact a generalization of traditional pre- and post-conditions for methods as used in design-by-contract: a single terminal 'call-m' can appear in multiple productions, each of which is followed by a different assertion. Hence different preconditions (or post-conditions) can be used for the same method, depending on the context (grammar production) in which the event corresponding to the method call/return appears.

Attribute grammars in combination with assertions cannot express protocol that depend on data. Such protocols are common, for instance, the method `next` of an `Iterator` may not be called if directly `hasNext` was called directly before and *returns* **false**. To express protocols depending on data we consider attribute grammars enriched by *conditional productions* [18]. In such grammars, a production is chosen only when the given condition (a **boolean** expression over the inherited attributes) for that production is true. Hence conditions are evaluated before any of the symbols in the production are parsed, before synthesized attributes of the non-terminals appearing in the production are set and before assertions are evaluated. In contrast to assertions, conditions in productions affect the parsing process. The Worker.g grammar in the case study contains a conditional production for the 'T' non-terminal.

2.3 Discussion

We now briefly motivate our choice of attribute grammars extended by assertions as specifications and discuss its advantages over alternative formalisms.

Instead of context-free grammars, we could have selected push-down automata to specify protocol properties (formally these have the same expressive power). Unfortunately push-down automata cannot handle attributes. An extension of push-down automata with attributes results in a register machine. From a user perspective, the declarative nature and higher abstraction level of grammars (compared to the imperative and low-level nature of automata) makes them much more suitable than automata as a *specification* language. In fact, a push-down automaton which recognizes the same language as a given grammar is an *implementation* of a parser for that grammar.

Both the BufferedReader above and the case study use only regular grammars. Since regular grammars simplify parsing compared to context-free grammars, the question arises if we can reasonably restrict to regular grammars. Unfortunately this rules out many real-life use cases. For instance, the following grammar in EBNF specifies the valid protocol behavior of a stack:

$$S ::= (push\ S\ pop\ ?)*$$

It is well known that the language generated by the above grammar is not regular, so regular grammars (without attributes) cannot be used to enforce the safe use of a stack. It is possible to specify the stack using an attribute which counts the number of pushes and pops:

$$\begin{array}{lll} S ::= S_1\ push & (S.cnt = S_1.cnt + 1) & \\ \quad \mid \quad S_1\ pop & (S.cnt = S_1.cnt - 1)\{assert\ S.cnt >= 0;\} & \\ \quad \mid \quad \epsilon & (S.cnt = 0) & \end{array}$$

The resulting grammar is clearly less elegant and less readable: essentially it encodes (instead of directly expresses, as in the grammar above) a protocol-oriented property as a data-oriented one. The same problem arises when using regular grammars to specify programs with recursive methods. Thus, although theoretically possible, we do not restrict to regular grammars for practical purposes.

Ultimately the goal of run-time checking safety properties is to prevent unsafe ongoing behavior. To do so, errors must be detected as soon as they occur and the monitor must *immediately* terminate the system: it cannot wait until the program ends to detect errors. In other words, the monitor must decide *after every event* whether the current history is still valid. The simplest notion of a valid history (one which should not generate any error) is that of a word generated by the grammar. One way of fulfilling the above requirement, assuming this notion of validity, is to restrict to prefix-closed grammars. Unfortunately, it is not possible to decide whether a context-free grammar is prefix-closed. The following lemmas formalize this result:

Lemma 1. *Let L_M be the set of all accepting computation histories*[3] *of a Turing Machine M. Then the complement $\overline{L_M}$ is a context-free language.*

Proof. See [20].

Lemma 2. *It is undecidable whether a context-free language is prefix-closed.*

Proof. We show how the halting problem for M (which is undecidable) can be reduced to deciding prefix-closure of $\overline{L_M}$. To that end, we distinguish two cases:

1. M does not halt. Then L_M is empty so $\overline{L_M}$ is universal and hence prefix-closed.

[3] A computation history of a Turing Machine is a sequence $C_0\#C_1\#C_2\#\ldots$ of configurations C_i. Each configuration is a triple consisting of the current tape contents, state and position of the read/write head. Due to a technicality, the configurations with an odd index must actually be encoded in reverse.

2. M halts. Then there is an accepting history $h \in L_M$ (and $h \notin \overline{L_M}$). Extend h with an illegal move (one not permitted by M) to the configuration C, resulting in the history $h\#C$. Clearly $h\#C$ is not a valid accepting history, so $h\#C \in \overline{L_M}$. But since $h \notin \overline{L_M}$, $\overline{L_M}$ is not prefix-closed.

Summarizing, M halts if and only if $\overline{L_M}$ is not prefix-closed. Thus if we could decide prefix-closure of the context-free language (lemma 1) $\overline{L_M}$, we could decide whether M halts.

Since prefix-closure is not a decidable property of grammars (not even if they don't contain attributes) we propose the following alternative definition for the valid histories. A communication history is valid if and only if it and all its prefixes are generated by the grammar. Note that this new definition naturally fulfills the above requirement of detecing errors after every event. And furthermore, this notion of validity is decidable assuming the assertions used in the grammar are decidable. As an example of this new notion of validity, consider the following modification of the above grammar:

$$
\begin{array}{lll}
T ::= S & & \{assert\ S.cnt >= 0;\} \\
S ::= S_1\ push & & (S.cnt = S_1.cnt + 1) \\
\quad\ \mid\ \ S_1\ pop & & (S.cnt = S_1.cnt - 1) \\
\quad\ \mid\ \ \epsilon & & (S.cnt = 0)
\end{array}
$$

Note that the history *push pop* is a word generated by this grammar, but not its prefix *pop*, which as such will generate an error (as required). Note that thus in general invalid histories are guaranteed to generate errors. On the other hand, if a history generates an error all its extensions are therefore also invalid.

Observe that our approach monitors only safety properties ('prevent bad behavior'), not liveness ('something good eventually happens'). This restriction is not specific to our approach: liveness properties in general cannot be rejected on any finite prefix of an execution, and monitoring only checks finite prefixes for violations of the specification. Most liveness properties fall in the class of the non-monitorable properties [2, 19]. However it *is* possible to ensure liveness properties for terminating programs: they can then be reformulated as safety properties. For instance, suppose we want to guarantee that a method `void m()` is called before the program ends. Introduce the following global view:

```
global view livenessM {
  call void C.m() m,
  return static void C.main(String[]) main
}
```

The occurence of the 'main' event (i.e., a return of the main method of the program) signifies the program is about to terminate. Define the EBNF grammar $S ::= \epsilon \mid m \mid m+\ main$

(where '+' stands for one or more repetitions). This grammar achieves the desired effect since the only terminating executions allowed are those containing `m`. In local views a similar effect is obtained by including the method `finalize` instead of `main`.

3 Tool Architecture

In this section we describe the tool architecture of the run-time assertion checker. The checker integrates four different components: a state-based assertion checker, a parser generator, a monitoring tool, and a general tool for meta-programming. These components are traditionally used for very diverse purposes and normally do not need to interact with each other. We investigate requirements needed to achieve a seamless integration, motivated by describing the workflow of the run-time checker. Finally, we instantiate the components with actual tools and evaluate them.

3.1 Workflow

A user starts executing a Java class with a main statement. Suppose that during execution, a method listed in a communication view is called. The history should be updated to reflect the addition of the method call. Thus the question arises how to represent the history. A **meta-program** generates for each message in the communication view a class (subsequently called 'token classes') containing the following fields: the object identitities of the *caller* and *callee*, the actual parameter values, and for return messages additionally a field `result` to store the return value. The history can then be represented as a Java `List` of instances of token classes.

Next, the **monitoring tool** should update the history whenever a call or return listed in a view occurs. Thus the monitoring tool should be capable of executing user-defined code directly before method calls and directly after method returns. Moreover, it must be able to read the identity of the callee, caller, and parameters/return-value.

After the history is updated the SAGA must decide whether it still satisfies the specification (the attribute grammar). Observe that a communication history can be seen as a sequence of tokens (in our setting: communication events). Since the attribute grammar together with the assertions generate the language of all valid histories, checking whether a history satisfies the specification reduces to deciding whether the history can be parsed by a parser for the attribute grammar, where moreover during parsing the assertions must evaluate to true.

Therefore the **parser generator** creates a parser for the given attribute grammar. Since the history is a list of token class objects, the parser must support parsing streams of user-defined token types. As the (user-defined) attributes of non-terminals in the grammar are defined in terms of built-in attributes of terminals (recall those are for example, actual parameter values), and clearly the built-in attributes are Java objects, the user-defined attributes must also be

Java objects. Consequently the target language for the parser generator must be Java, and it must support executing user-defined Java code to define the attribute value in rule actions. The use of Java code to define attribute values ensures they are computable. Furthermore, assertions are allowed in-between any two (non)-terminals, thus the parser generator should support user-defined actions between arbitrary grammar symbols. Once the parser is generated, it is triggered whenever the history of an object is updated.

During parsing, the **state-based assertion checker** proceeds to evaluate the assertions in the grammar on the newly computed attribute values. The result is either a parse or assertion error, which indicates that the current communication history has violated the specification in the attribute grammar, or a parse tree with new attribute values.

3.2 Implementation

In this section we instantiate each of the four different components (meta-programming, monitoring tool, parser generator, and state-based run-time assertion checker) with a state-of-the art tool. We report on our experiences with the particular tools and discuss the extent to which the previously formulated requirements are fulfilled.

Rascal [13] is a powerful tool-supported meta-programming language tailored for program analysis, program transformation, and code generation. We have written a Rascal program of approximately 600 lines in total which generates the token class for each message in the view, and generates glue code to trigger the AspectJ and parser at the appropriate times. Overall our experience with Rascal was quite positive: its powerful parsing, pattern matching, and transforming concrete syntax features were indispensable in the implementation of SAGA.

As the parser generator we tested ANTLR [17], a state-of-the-art parser generator. It generates fast recursive descent parsers for Java, has direct support for both synthesized and inherited attributes, it supports grammars in EBNF form and most importantly allows a custom stream of token classes. It even supports *conditional productions*: such productions are only taken during parsing whenever an associated Boolean expression (the condition) is true. Attribute grammars with conditional productions express protocols that depend on data, and typically are not context-free. The worst-case time complexity any parser ANTLR generates is quadratic in the number of tokens to parse. The main drawbacks of ANTLR are that it can only handle LL(*) grammars[4], and its lack of support for incremental parsing, though support for incremental is planned by the ANTLR developers. An incremental parser computes a parse tree for the new history based on the parse trees for prefixes of the history. In our setting, since the attribute grammar specifies invariant properties of the ongoing behavior, a new parse tree is computed after each call/return, hence parse trees for all prefixes of the current history can be exploited for incremental parsing [11].

[4] A strict subset of the context-free grammars. Left-recursive grammars are not LL(*). A precise definition can be found in [17].

We have not been able to find any Java parser generator which supported general context-free grammars and incremental parsing of attribute grammars.

We have tested two state-based assertion languages: standard Java assertions and the Java Modeling Language (JML). Both languages suffice for our purposes. JML is far more expressive than the standard Java assertions, though its tool support is not ready for industrial usage. In particular, the last stable version of the JML run-time assertion checker dates back over 8 years, when for example generics were not supported yet. The main reason is that JML's run-time assertion checker only works with a proprietary implementation of the Java compiler, and unsurprisingly it is costly to update the propietary compiler each time the standard compiler is updated. This problem is recognized by the JML developers [4]. OpenJML[5], a new pre-alpha version of the JML run-time assertion checker integrates into the standard Java compiler, and initial tests with it provided many valuable input for real industrial size applications. See the Sourceforge tracker for the kind of issues we have encountered when using OpenJML.

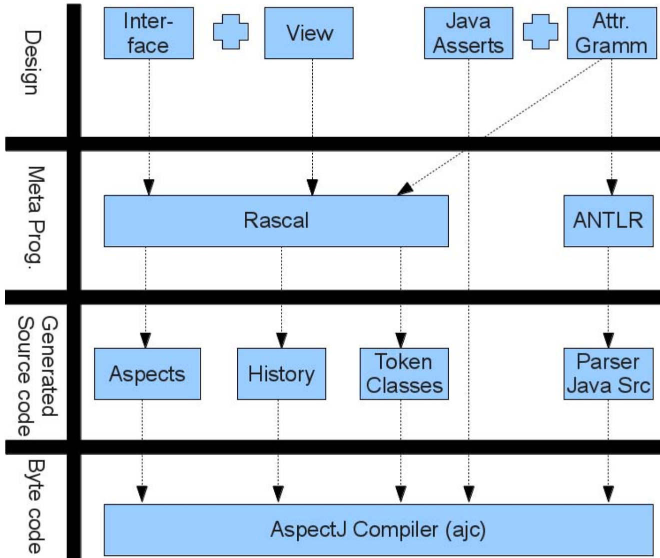


Fig. 5. SAGA Tool Architecture

Aspects. AspectJ is tailored for monitoring. It can intercept method calls and returns conveniently with pointcuts, and weave in user-defined code (advices) which is executed before or after the intercepted call. In our case the pointcuts correspond to the calls and returns of the messages listed in the communication view. The advice consists of code which updates the history. The code for the aspect is generated from the communication view automatically by the Rascal

[5] `jmlspecs.sourceforge.net`

meta-program. Advice is woven into Java source code, byte code, or at class load-time fully automatically by AspectJ. We use the inter-type declarations of AspectJ to store the local history of an object as a field in the object itself. This ensures that whenever the object goes out of scope, so does its history and consequently reduces memory usage. Clearly the same does not hold for global histories, which are stored inside a separate Aspect class. Figure 6 shows a generated aspect. The second and third lines specify the relevant method. The fourth line binds variables ('clr', 'cle', ...) to the appropriate objects. The fifth line ensures that the aspect is applied only when Java assertions are turned on. Assertions can be turned on or off for each communication view individually. The fifth line contains the advice that updates the history. Note that since the event came was defined in a local view, the history is treated as a field of the callee (and will not persist in the program indefinitely but rather is garbage collected as soon as callee object itself is).

```
    /* call int read(char[] cbuf, int off, int len); */
before(Object clr, BufferedReader cle, char[] cbuf, int off, in len):
 (call( int *.read(char[], int, int))
  && this(clr) && target(cle) && args(cbuf, off, len)
  && if(BReaderHistoryAspect.class.desiredAssertionStatus() )) {
    cle.h.update(new call_push(clr, cle, cbuf, off, len));
}
```

Fig. 6. Aspect for the event 'call int read(char[] cbuf, int off, int len)'

We have investigated two alternatives for the monitoring component not based on aspect-oriented programming: Rascal and Sun's implementation of the Java Debugging Interface. With Rascal one can weave advice by defining a transformation on the actual Java source code of the program to test. This requires a full Java grammar (which must be kept in sync with the latest updates to Java). To capture the identity of the callee, parameter values and return value of a method, one only needs to transform that particular method (i.e., locally). But inside the method there is no way to access the identity of the caller. Java does offer facilities to inspect stack frames, but these frames contain only static entities, such as the name of the method which called the currently executing method, or the type of the caller, but not the caller itself. To capture the caller, a global transformation at all call-sites is needed (and in particular one needs to have access to the source code of all clients which call the method). The same problem arises in monitoring calls to required methods. Finally, it proved to quickly get very complex to handle all Java features. We wrote an initial version of a weaver in Rascal which already took over 150 lines (over half of the full checker at the time) without supporting method calls appearing inside expressions, inheritance and dynamic binding. This approach is also unsuitable for

black-box testing where only byte code is available (limiting the applicability of the tool). In summary, it is possible to implement monitoring by defining a code transformation in Rascal, but this rules out black-box testing and quickly gets complex due to the need for a full (up to date) Java grammar and the complexity of the full Java language.

The Sun debugger is part of the standard Java Development Kit, hence maintenance of the debugger is practically guaranteed. The debugger starts the original user program in a separate virtual machine which is monitored for occurences of `MethodEntryEvent` (method calls) and `MethodExitEvent` (method returns). Whenever such an event occurs the debugger can execute an event handler. However accessing the values of the parameters and return value of events is difficult, one has to use low-level `StackFrames`. As a major disadvantage, we found that the debugger is very slow (an order of magnitude slower than AspectJ), in fact it was responsible for the majority of the overhead of the run-time checker. Finally, in contrast to AspectJ it not possible to add fields to objects, thus local histories never go out of scope, even if the object itself is already long destroyed.

In summary, the use of aspect-oriented programming greatly improved performance compared to the debugger-based solution and was much simpler than implementing our own weaver with code transformations, especially to handle intricate language features.

4 Case Study

Fredhopper provides the Fredhopper Access Server (FAS). It is a distributed concurrent object-oriented system that provides search and merchandising services to eCommerce companies. Briefly, FAS provides to its clients structured search capabilities within the client's data. Each FAS installation is deployed to a customer according to the FAS deployment architecture (See Figure 7).

FAS consists of a set of live environments and a single staging environment. A live environment processes queries from client web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the *Replication Protocol*. The Replication Protocol is implemented by the *Replication System*. The Replication System consists of a *SyncServer* at the staging environment and one *SyncClient* for each live environment. The SyncServer determines the schedule of replication, as well as its content, while SyncClient receives data and configuration updates according to the schedule.

Replication Protocol

The SyncServer communicates to SyncClients by creating *Worker* objects. Workers serve as the interface to the server-side of the Replication Protocol. On the other hand, SyncClients schedule and create *ClientJob* objects to handle communications to the client-side of the Replication Protocol. When transferring data

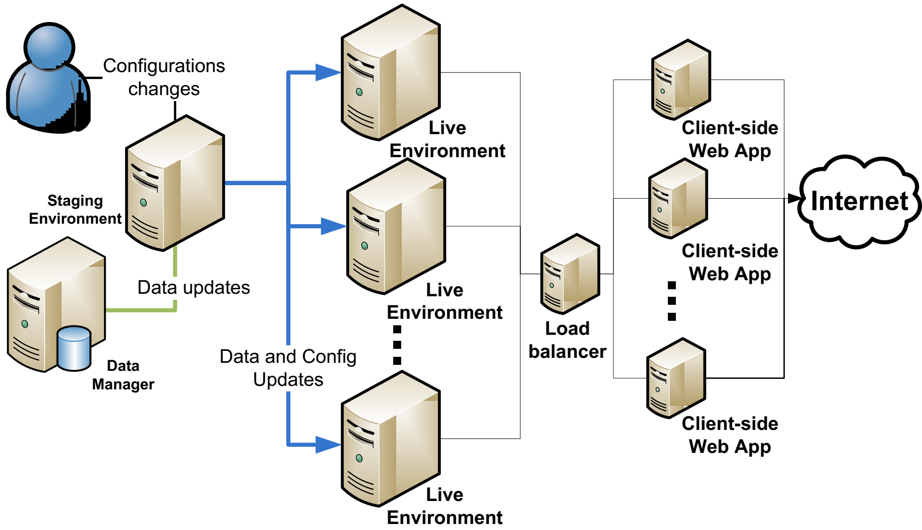


Fig. 7. An example FAS deployment

between the staging and the live environments, it is important that the data remains *immutable*. To ensure immutability without interfering the read and write accesses of the staging environment's underlying file system, the SyncServer creates a *Snapshot* object that encapsulates a snapshot of the necessary part of the staging environment's file system, and periodically *refreshes* it against the file system. This ensures that data remains immutable until it is deemed safe to modify it. The SyncServer uses a *Coordinator* object to determine the safe state in which the Snapshot can be refreshed. Figure 8 shows a UML sequence diagram concerning parts of the replication protocol with the interaction between a SyncClient, a ClientJob, a Worker, a SyncServer, a Coordinator, and a Snapshot. The diagram also shows a *Util* class that provides static methods for writing to and reading from `Stream`. The figure assumes that SyncClient has already established connection with a SyncServer and shows how a ClientJob from the SyncClient and a Worker from a SyncServer are instantiated for interaction. For the purpose of this paper we consider this part of the Replication Protocol as a *replication session*.

4.1 Specification

In this section we show how to modularly decompose object interaction behavior depicted by the UML sequence diagram in Figure 8 using SAGA. Figure 9 shows the corresponding interfaces and classes, note that we do not consider SyncClient as our interest is in object interactions of a replication session, that is after `ClientJob.start()` has been invoked.

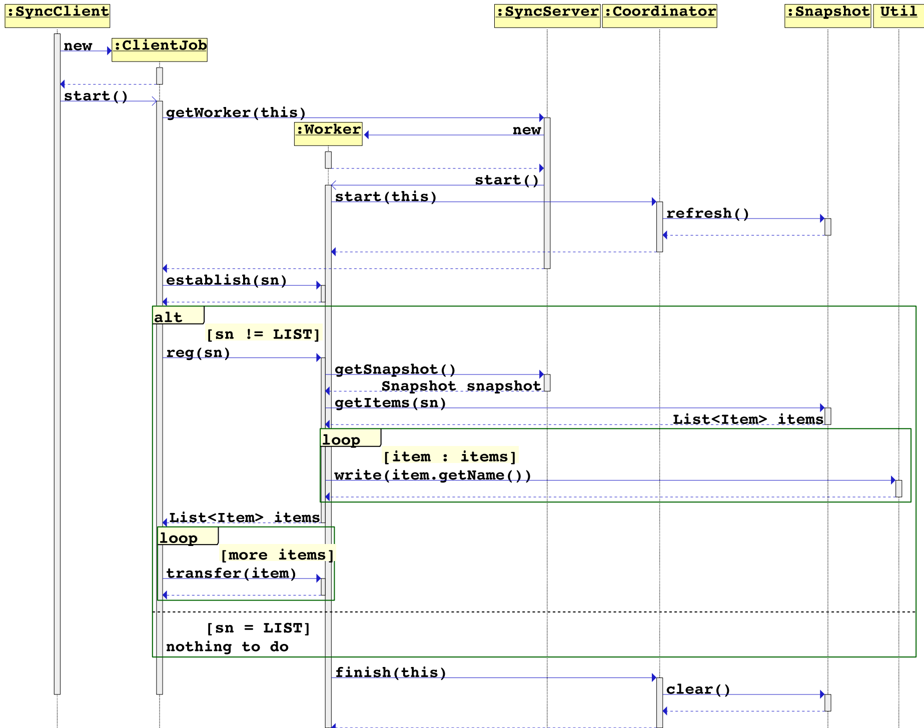


Fig. 8. Replication interaction

The protocol descriptions and specifications considered in this case study have been obtained by manually examining the behavior of the existing implementation, by formalizing available informal documentations, and by consulting existing developers on intended behavior. Here we first provide such informal descriptions of the relevant object interactions:

– Snapshot: at the initialization of the Replication System, `refresh` should be called first to refresh the snapshot. Subsequently the invocations of methods `refresh` and `clear` should alternate.
– Coordinator: neither of methods `start` and `finish` may be invoked twice in a row with the same argument, and method `start` must be invoked before `finish` with the same argument can be invoked.
– Worker: `establish` must be called first. Furthermore, `reg` may be called *if* the input argument of `establish` is not "LIST" but the name of a specific replication schedule, and that `reg` must take that name as an input argument. When the `reg` method is invoked and before the method returns, the Worker must obtain the replication items for that specific replication schedule via method `items` of the Snapshot object. The Snapshot object must be obtained via method `snapshot` of its SyncServer, which must be obtained via the method `server`. It must notify the name of each replication item to

```
interface Snapshot {
 void refresh();
 void clear();
 List<Item> items(String sn);
}

interface Worker {
 void establish(String sn);
 List<Item> reg(String sn);
 void transfer(Item item);
 SyncServer server();
}
```

```
interface SyncServer {
 Snapshot snapshot();
}

interface Coordinator {
 void start(Worker t);
 void finish(Worker t);
}

class Util {
 static void write(String s) { .. }
}
```

Fig. 9. Interfaces of Replication System

its interacting SyncClient. This notification behavior is implemented by the static method `write` of the class `Util`. The method `reg` also checks for the validity of each replication item and so the method must return a subset of the items provided by the method `items`. Finally `transfer` may be invoked after `reg`, one or more times, each time with a unique replication item, of type `Item`, from the list of replication items, of type `List<Item>`, returned from `reg`.

Figure 10 specifies communication views. They provide partial mappings from message types (method calls and returns) that are local to individual objects to grammar terminal symbols. Note that the specification of the Worker's behavior is modularly captured by two views: `WorkerHistory` and `WorkerRegHistory`. The view `WorkerHistory` exposes methods `establish`, `reg`, and `transfer`. Using this view we would like to capture the overall valid interaction in which Worker is the callee of methods, and at the same time the view helps abstracting away the implementation detail of individual methods. The view `WorkerRegHistory`, on the other hand, captures the behavior inside `reg`. According to the informal description above, the view projects incoming method calls and returns of `reg`, outgoing method calls to `server` and `items`, as well as the outgoing static method calls to `write`.

We now define the abstract behavior of the communication views, that is, the set of allowable sequences of interactions of objects restricted to those method calls and returns mapped in the views. Each local view also defines the file containing the attribute grammar, whose terminal symbols the view maps method invocations and returns to. Specifically, Figure 11 shows the attribute grammars `Snapshot.g`, `Coordinator.g`, `Worker.g` and `WorkerReg.g` for views `SnapshotHistory`, `CoordinatorHistory`, `WorkerHistory` and `WorkerRegHistory` respectively.

```
local view SnapshotHistory
grammar Snapshot.g
specifies Snapshot {
 call void refresh() rf,
 call void clear() cl
}
```

```
local view CoordinatorHistory
grammar Coordinator.g
specifies Coordinator {
 call void start(Worker t) st,
 call void finish(Worker t) fn
}
```

```
local view WorkerHistory grammar Worker.g
specifies Worker {
 call void establish(String sn) et,
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 call void transfer(Item item) tr
}
```

```
local view WorkerRegHistory grammar WorkerReg.g
specifies Worker {
 call List<Item> reg(String sn) rg,
 return List<Item> reg(String sn) is,
 return Snapshot SyncServer.snapshot() sp,
 call List<Item> Snapshot.items(String sn) ls,
 return List<Item Snapshot.items(String sn) li,
 call static void Util.write(String s) wr
}
```

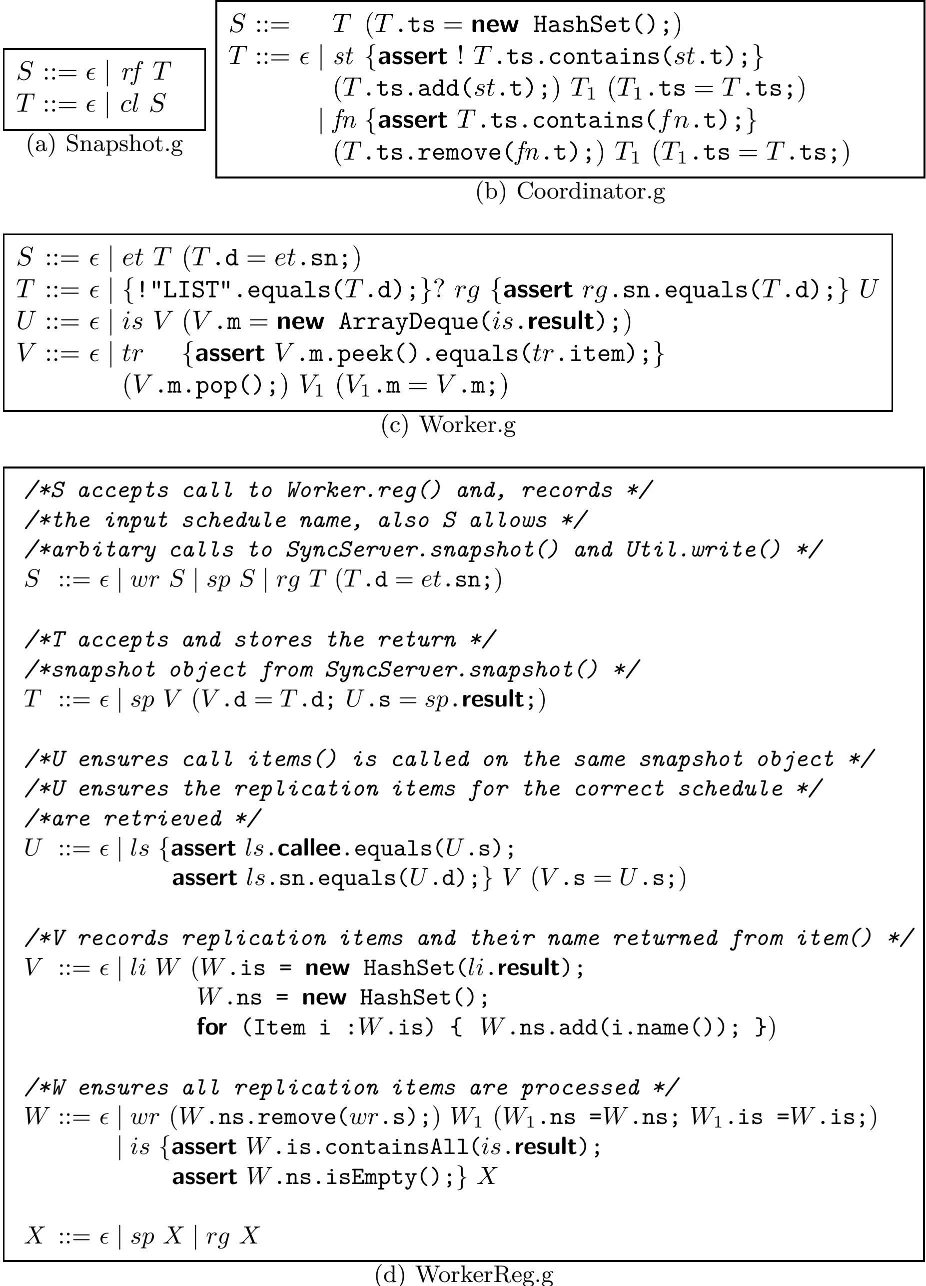
Fig. 10. Communication Views

The simplest grammar `Snapshot.g` specifies the interaction protocol of Snapshot. It focuses on invocations of methods `refresh` and `clear` per Snapshot object. The grammar essentially specifies the regular expression (`refresh clear`)$*$.

The grammar `Coordinator.g` specifies the interaction protocol of Coordinator. It focuses on invocations of methods `start` and `finish`, both of which take a Worker object as the input parameter. These method calls are mapped to terminal symbols *st* and fn, while their inherited attribute is a `HashSet`, recording the input parameters, thereby enforcing that for each unique Worker object as an input parameter only the set of sequences of method invocations defined by the reqular expression (`start finish`)$*$ is allowed.

The grammar `Worker.g` specifies the interaction protocol of Worker It focuses on invocations and returns of methods `establish`, `reg` and `transfer`. The grammar specifies that for each Worker object, `establish` must be first invoked, then followed by `reg`, and then zero or more `transfer`, that is, the regular expression (`establish reg transfer`$*$). We use the attribute definition of the grammar to ensure the following:

- The input argument of `establish` and `reg` must be the same;
- `reg` can only be invoked if the input argument of `establish` is not “LIST”;



```
S ::= ε | rf T
T ::= ε | cl S
```

(a) Snapshot.g

```
S ::=     T (T.ts = new HashSet();)
T ::= ε | st {assert ! T.ts.contains(st.t);}
          (T.ts.add(st.t);) T1 (T1.ts = T.ts;)
        | fn {assert T.ts.contains(fn.t);}
          (T.ts.remove(fn.t);) T1 (T1.ts = T.ts;)
```

(b) Coordinator.g

```
S ::= ε | et T (T.d = et.sn;)
T ::= ε | {!"LIST".equals(T.d);}? rg {assert rg.sn.equals(T.d);} U
U ::= ε | is V (V.m = new ArrayDeque(is.result);)
V ::= ε | tr    {assert V.m.peek().equals(tr.item);}
          (V.m.pop();) V1 (V1.m = V.m;)
```

(c) Worker.g

```
/*S accepts call to Worker.reg() and, records */
/*the input schedule name, also S allows */
/*arbitary calls to SyncServer.snapshot() and Util.write() */
S  ::= ε | wr S | sp S | rg T (T.d = et.sn;)

/*T accepts and stores the return */
/*snapshot object from SyncServer.snapshot() */
T  ::= ε | sp V (V.d = T.d; U.s = sp.result;)

/*U ensures call items() is called on the same snapshot object */
/*U ensures the replication items for the correct schedule */
/*are retrieved */
U  ::= ε | ls {assert ls.callee.equals(U.s);
              assert ls.sn.equals(U.d);} V (V.s = U.s;)

/*V records replication items and their name returned from item() */
V  ::= ε | li W (W.is = new HashSet(li.result);
                W.ns = new HashSet();
                for (Item i :W.is) { W.ns.add(i.name()); })

/*W ensures all replication items are processed */
W ::= ε | wr (W.ns.remove(wr.s);) W1 (W1.ns =W.ns; W1.is =W.is;)
        | is {assert W.is.containsAll(is.result);
              assert W.ns.isEmpty();} X

X  ::= ε | sp X | rg X
```

(d) WorkerReg.g

Fig. 11. Attribute Grammars

– The return value of `reg` is a list of `Item` objects such that `transfer` is invoked with each of `Item` in that list from position 0 to the size of that list.

The grammar `WorkerReg.g` specifies the behavior of the method `reg` of Worker. It focuses on the invocations and returns of method `reg` of Worker as well as the outgoing method calls and returns of `Util.write` and `SyncServer.snapshot` and `Snapshot.items`. At the protocol level the grammar specifies the regular expression (`snapshot items write`$*$) inside the invocation method `reg`. We use attribute definition to ensure the following:

– `Snapshot.items` must be called with the input argument of `reg` and it must be called on the Snapshot object that is identical to the return value of `SyncServer.snapshot`;
– The static method `Util.write` must be invoked with the value of `Item.name` for each `Item` object in the Collection returned from `Snapshot.items`;
– The returned list of `Item` objects from `reg` must be a subset of that returned from `Snapshot.items`.

Notice that methods `Util.write` and `SyncServer.snapshot` may be invoked outside of the method `reg`. However, this particular behavioral property does not specify the protocol for those invocations. The grammar therefore abstracts from these invocations by allowing any number of calls to `Util.write` and `SyncServer.snapshot` before and after `reg`.

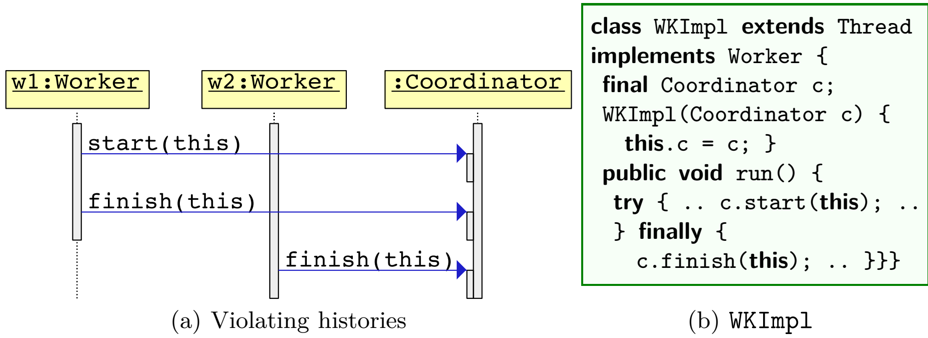
4.2 Experiment

We applied SAGA to the Replication System. The current Java implementation of FAS has over 150,000 lines of code, and the Replication System has approximately 6400 lines of code, 44 classes, and 5 interfaces.

We have successfully integrated the SAGA into the quality assurance process at Fredhopper. The quality assurance process includes automated testing that includes automated unit, integration, and system tests as well as manual acceptance tests. In particular system tests are executed twice a day on instances of FAS on a server farm. Two types of system tests are scenario and functional testing. Scenario testing executes a set of programs that emulate a user and interact with the system in predefined sequences of steps (scenarios). At each step they perform a configuration change or a query to FAS, make assertions about the response from the query, etc. Functional testing executes sequences of queries, where each query-response pair is used to decide on the next query and the assertion to make about the response. Both types of tests require a running FAS instance and as a result we may leverage SAGA by augmenting these two automated test facilities with run-time assertion checking using SAGA.

To integrate of SAGA with the system tests, we employ Apache Maven tool[6], an open source Java-based tool for managing dependencies between applications and for building dependency artifacts. Maven consists of a project object model

[6] `maven.apache.org`



(a) Violating histories (b) `WKImpl`

Fig. 12. Incorrect behavior

(POM), a set of standards, a project lifecycle, and an extensible dependency management and build system via plug-ins. We use its build system to automatically generate and package the parser/lexer of attribute grammars as well as aspects from views and grammars. We expose the packaged aspects, parser, and lexer to FAS instance on the server farm and employ Aspectj using load-time weaver for monitoring method calls/returns during the execution of FAS instances on the server farm. Table 2 shows the number of join point matches during the execution of 766 replication sessions over live client data. Figure 13 shows the exection time of the 766 replication sessions with and without the integration of SAGA in milliseconds. Despite the fact that we cannot control the exact flow of control of the replication sessions (due to dependence on user input), the graph clearly shows that the integration of SAGA has minimal performance impact on the execution time.

Table 2. Join point matches in 766 replication sessions

Join point	Terminal	Match
call static `write`	wr	247446
return `snapshot`	sp	3061
call `transferItem`	tr	1101
return `reg` (`WorkerHistory`)	is	765
return `reg` (`WorkerRegHistory`)	is	765
call `establish`	et	766
call `reg` (`WorkerHistory`)	rg	765
call `reg` (`WorkerRegHistory`)	rg	765
return `items`	li	765
call `start`	st	766
call `finish`	fn	766
call `items`	ls	765
call `refresh`	rf	766
call `clear`	cl	766

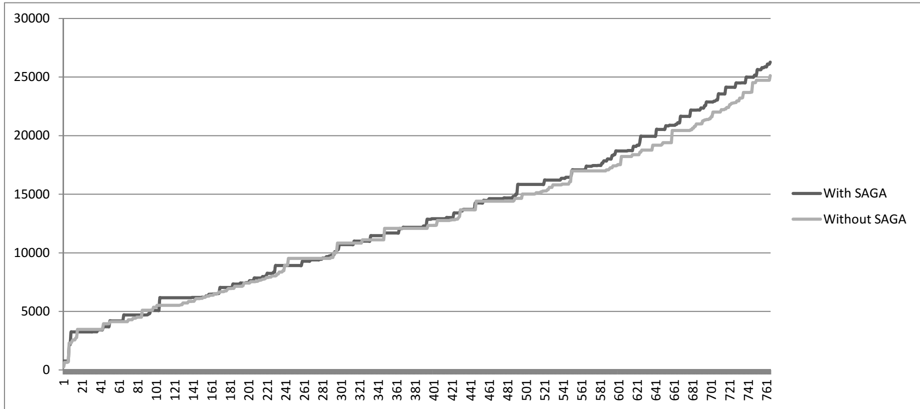


Fig. 13. Comparison of the execution time (milliseconds) of the replication sessions with and without the integration of SAGA

During this session we have found an assertion error at join point **call** `finish` due to the condition T`.ts.contains(`fn`.t)` not being satisfied at non-terminal T of the grammar Coordinator.g. Specifically, the implementation of Worker (`WKImpl`) that invoke `finish` before `start`. Figure 12(a) shows the sequence diagram automatically generated from the output of SAGA on the invalid histories causing the assertion error. Figure 12(b) shows part of the implementation of `WKImpl`. It turns out that in the `run` method of `WKImpl`, the method `start` is invoked inside a **try** block while the method `finish` is invoked in the corresponding **finally** block. As a result when there is an exception being thrown by the execution preceding the invocation of `start` inside the **try** block, for example a network disruption, `finish` would be invoked without `start` being invoked.

5 Conclusion

We developed SAGA, a run-time checker which fully automatically checks properties of both the protocol behavior and data-flow of message sequences in a declarative manner. We identified the different components of SAGA and evaluated SAGA on an industrial case study of the eCommerce company Fredhopper. The results of this case study show the feasability of our method for run-time verification in industrial practice (it has already led to the integration of SAGA into the software lifecycle at Fredhopper), in contrast to methods for static verification which require both an in-depth knowledge of the case study and the underlying theorem prover. A beta version of SAGA can be found on `http://www.cwi.nl/~cdegouw`.

Related Work. A preliminary version of a prototype of our tool containing some of the basic underlying ideas was presented at the workshop Formal Techniques

for Java-Like Programs 2010 and appeared in its *informal* proceedings[7]. In the current paper we apply and evaluate a new version to an industrial case study and succesfully integrate SAGA into the quality assurance process of Fredhopper. Based on this application and evaluation we extended our framework to support a more general class of grammars to specify data-dependent protocol behavior. Furthermore, The new version features a tighter integration of attribute grammars and assertions. Finally the support for the features listed in Table 1 is new.

There exist many other interesting approaches to monitoring message sequences, none of which address their integration with the general context of run-time assertion checking. Consequently, all the other approaches only allow a combination of a very restricted class of data-oriented properties and protocol properties. For example, Martin et al. [15] introduce a Program Query Language (PQL) for detecting errors in sequences of communication events. PQL was updated last in 2006 and does not support user-defined properties of data. Allan et al. [1] develop an extension of AspectJ with a history-based language feature called Tracematches that enables the programmer to trigger the execution of extra code by specifying a regular pattern of events in a computation trace. The underlying pattern matching involves a binding of values to free variables. Nobakht et al. [16] monitors calls and returns with the same Java Debugger Architecture we have also evaluated in the implementation section. The debugger is very slow compared to aspect-oriented approaches. Their specification language is equivalent in expressive power to regular expressions. Because the grammar for the specifications is fixed, the user cannot specify a convenient structure themselves, and data is not considered. Chen et al. [5] present JavaMOP, a run-time monitoring tool based on aspect-oriented programming which uses context-free grammars to describe properties of the control flow of histories. However, properties on the data-flow are *predefined* built-in functions (basically AspectJ functions such as a 'target' to bind the callee and 'this' to bind the caller, comparable to built-in attributes of terminals in our setting). This limits the expression of data properties. Though, to circumvent this limitation one may hack general properties into the tool implementation. In contrast, our approach supports a general methodology to introduce systematically *user-defined* properties, by means of attributes of non-terminals. Furthermore, SAGA supports conditional productions which are essential to specify protocols dependent on data in a declarative manner. Finally, JavaMOP does not directly support the specification of local histories (i.e., monitoring the messages sent and received by a single object). LARVA is developed by Colombo et al. [8]. The specification language has an imperative flavor: users define a finite state machine to define the allowed history (i.e., one has to 'implement' a regular expression themselves). It is not possible to directly express context-free protocols. Data properties are supported, though in a limited manner, by enriching the state machine with conditions on method parameters or return values. It is not possible to specify a local

[7] Available in the ACM Digital Library with the title "Prototyping a tool environment for run-time assertion checking in JML with communication histories", authored by Frank S. de Boer, Stijn de Gouw and Jurgen Vinju

history of a single object. DeLine and Fähndrich [10] propose a statically checkable typestate system for object-oriented programs. Typestate specifications of protocols correspond to finite state machines (assertions are not considered in their approach), thus for example a stack cannot be properly specified.

To the best of our knowledge, no other approach *integrates* protocol-oriented properties into existing state-based assertion languages. The integration does not involve an extension of the syntax and semantics of the assertion language itself. As an important consequence, no change in the implementation of the state-based assertion checker is needed, in contrast to the following works. Cheon and Perumandla present in [6] an extension of the JML *compiler* with call sequence assertions. Call sequence assertions are regular expressions (proper context-free grammars cannot be handled) over method names and the data sent in calls and returns is not considered. Protocol properties (call sequence assertions) are handled separately from data properties, and as such are not integrated into the general context of (data) assertions. The proposed extension to call sequence assertions involves changing the existing JML-compiler (in particular, both the syntax and the semantics of JML assertions are extended), whereas in our test suite integrating with JML consists only of a simple pre-processing stage. Consequently in our approach no change in the JML-compiler is needed, and new versions of the JML-compiler are supported automatically, as long as they are backwards compatible. Hurlin [12] presents an extension of the previous work to handle multi-threading, which however is not supported by run-time verification (instead it discusses static verification). As in the previous work, an integration of protocol properties with assertions is not considered. Trentelman and Huisman [21] describe a new formalism extending JML assertions with Temporal Logic operators. A translation for a subset of the Temporal Logic formulae back to standard JML is described, and as future work they intend to integrate their extension into the standard JML-grammar which requires a corresponding new compiler.

Future Work. SAGA visualizes the offending history of a Java program that violates the given attribute grammar in the form of a UML sequence diagram. For industrial applications the histories (and consequently the corresponding diagram) can get very large, even when projecting away irrelevant events with the communication view. In such cases we found it is very useful to (further) filter events from the diagram, focussing on a specific part of the diagram. For instance, only showing all events in which a particular object was involved. The sequence diagram editor used by SAGA provides preliminary support for filtering using low-level UNIX system-based utilities grep and sed, but more high-level solutions specifically tailored for sequence diagrams would be even more useful. Furthermore, for debug purposes it would be convenient to visualize the current contents of the heap.

Another line of future work concerns offline monitoring. Offline monitoring serializes and stores the history of a running program in a file. This file is checked later for correctness, possibly on a different computer. This allows companies to enable monitoring production code deployed at clients with little performance

penalty: the histories can be checked on dedicated computers at the company instead of at the client. A potential disadvantage of off line monitoring is that it is not possible anymore to stop a running system directly after the attribute grammar is violated (or inspect the content of the heap at that time).

Acknowledgements. We wish to express our gratitude to Behrooz Nobakht for his help on the integration with the Java debugger and Jurgen Vinju for the helpful discussions and major contributions to our Rascal tool.

References

1. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L.J., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. In: OOPSLA, pp. 345–364 (2005)
2. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
3. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
4. Chalin, P., James, P.R., Karabotsos, G.: JML4: Towards an industrial grade IVE for java and next generation research platform for JML. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 70–83. Springer, Heidelberg (2008)
5. Chen, F., Rosu, G.: MOP: an efficient and generic runtime verification framework. In: OOPSLA, pp. 569–588 (2007)
6. Cheon, Y., Perumandla, A.: Specifying and checking method call sequences of java programs. Software Quality Journal 15(1), 7–25 (2007)
7. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes 31(3), 25–37 (2006)
8. Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time Java programs (tool paper). In: SEFM, pp. 33–37 (2009)
9. de Boer, F.S., de Gouw, S.: Run-time verification of black-box components using behavioral specifications: An experience report on tool development. In: Păsăreanu, C.S., Salaün, G. (eds.) FACS 2012. LNCS, vol. 7684, pp. 128–133. Springer, Heidelberg (2013)
10. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)
11. Hedin, G.: Incremental attribute evaluation with side-effects. In: Hammer, D. (ed.) CCHSC 1988. LNCS, vol. 371, pp. 175–189. Springer, Heidelberg (1989)
12. Hurlin, C.: Specifying and checking protocols of multithreaded classes. In: ACM Symposium on Applied Computing (SAC 2009), pp. 587–592. ACM Press (2009)
13. Klint, P., van der Storm, T., Vinju, J.: Rascal: a domain specific language for source code analysis and manipulation. In: Walenstein, A., Schupp, S. (eds.) Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), pp. 168–177 (2009)
14. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory 2(2), 127–145 (1968)

15. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using PQL: a program query language. In: OOPLSLA (2005)
16. Nobakht, B., Bonsangue, M.M., de Boer, F.S., de Gouw, S.: Monitoring method call sequences using annotations. In: Barbosa, L.S., Lumpe, M. (eds.) FACS 2010. LNCS, vol. 6921, pp. 53–70. Springer, Heidelberg (2012)
17. Parr, T.: The Definitive ANTLR Reference. Pragmatic Bookshelf (2007)
18. Parr, T.J., Quong, R.W.: Adding semantic and syntactic predicates to LL(k): pred-LL(k). In: Fritzson, P.A. (ed.) CC 1994. LNCS, vol. 786, pp. 263–277. Springer, Heidelberg (1994)
19. Pnueli, A., Zaks, A.: PSL model checking and run-time verification via testers. In: FM, pp. 573–586 (2006)
20. Sipser, M.: Introduction to the theory of computation. PWS Publishing Company (1997)
21. Trentelman, K., Huisman, M.: Extending JML specifications with temporal logic. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 334–348. Springer, Heidelberg (2002)

Event Modules
Modularizing Domain-Specific Crosscutting RV Concerns

Somayeh Malakuti[1] and Mehmet Akşit[2]

[1] Software Technology Group, Technical University of Dresden, Germany
somayeh.malakuti@tu-dresden.de
[2] Software Engineering Group, University of Twente, The Netherlands
m.aksit@utwente.nl

Abstract. Runtime verification (RV) facilitates detecting the failures of software during its execution. Due to the complexity of RV techniques, there is an increasing interest in achieving abstractness, modularity, and compose-ability in their implementations by means of dedicated linguistic mechanisms. This paper defines a design space to evaluate the existing domain-specific languages for implementing RV techniques, and identifies the shortcomings of a representative set of these languages with respect to the design space. This paper advocates the need for a language composition framework, which offers the necessary mechanisms to achieve abstractness, modularity, and compose-ability in the implementation of domain-specific crosscutting concerns such as the concerns of RV techniques. We explain **event modules** as novel linguistic abstractions for modular implementation of domain-specific crosscutting concerns. This paper discusses the implementation of event modules in the EventReactor language, and illustrates the suitability of event modules to implement RV techniques by means of two complementary examples.

Keywords: runtime verification, domain-specific languages, aspect-orientation, event-based modularization, event-based composition.

1 Introduction

Runtime verification (RV) [1] aims at checking software against its desired properties while the software is executed, e.g., during testing or after it is deployed. Depending on the result of the verification process, various actions may be carried out such as notification, suspending execution, fault recovery, etc. In this paper, the term *RV technique* refers to the program that implements the functionality of the runtime verification, and the term *base software* refers to the software that is being verified by such a technique.

To apply RV techniques to real-world complex base software, there is a need for suitable implementation mechanisms/frameworks that ease the implementation of these techniques for industrial practitioners. One may argue that RV techniques can be implemented directly as an integral part of the base software in the same general-purpose language (GPL) as the base software. However, this requires programmers to have extensive knowledge about suitable algorithms and

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 27–69, 2014.
© Springer-Verlag Berlin Heidelberg 2014

mechanisms to implement RV techniques in a GPL. Moreover, the implementation of both base software and RV techniques can easily become complex and hard to comprehend. This is because RV techniques usually *crosscut* [2] the base software, meaning that they need to interact with various different parts of the base software to collect the necessary information and/or to heal them from failures.

The aforementioned problems in implementing RV techniques motivate language designers to seek suitable linguistic constructs for implementing RV techniques. A close look at the literature [3–14], lets us observe that achieving **abstractness**, **modularity**, and **compose-ability** is of interest in the implementation of RV techniques. The abstractness requirement is addressed by providing suitable domain-specific languages (DSLs) for implementing RV techniques. The modularity requirement is addressed by providing means to modularize individual concerns of RV techniques from each other and from the base software. The compose-ability requirement is addressed by providing suitable operators to compose individually modularized RV concerns with each other and with the base software under the specified constraints.

Although several RV DSLs have been introduced in the literature and this trend seems to be continuing, they fall short of fulfilling the abstractness, modularity, and compose-ability requirements. To be able to identify the source of their shortcomings, this paper identifies the concerns that typically appear in RV techniques, defines a design space for the RV DSLs, and evaluates a representative set of current RV DSLs with respect to this design space.

To overcome the identified shortcomings, this paper identifies characteristic features of RV techniques, and introduces **Event Composition Model**, which offers **event modules** as novel linguistic abstractions to achieve abstractness, modularity, and compose-ability in the implementation of domain-specific crosscutting concerns such as the concerns that exist in RV techniques. This paper explains **EventReactor** as a language composition framework that implements Event Composition Model, and by means of two examples, shows the suitability of EventReactor for implementing various kinds of RV techniques.

In this paper we extend our previous work [15, 16] in the following ways:

– We study a large number of RV DSLs and derive a conceptual model for RV techniques.
– We present a mind map of the design space for RV DSLs which facilitates the comparison of current RV DSLs.
– We evaluate a representative set of RV DSLs and identify their shortcomings in fulfilling abstractness, modularity, and compose-ability in the implementation.
– We define a new version of Event Composition Model, in which domain-specific concerns can be modularized and composed better.
– We present its implementation, EventReactor, which covers the design space.
– We demonstrate the suitability of EventReactor to achieve abstractness, modularity, and compose-ability in the implementation by means of two comprehensive examples.
– We present our evaluation of the runtime overhead of EventReactor.

This paper is organized as follows: Section 2 elaborates on the problem statement; Section 3 discusses the requirements for an RV language composition framework and explains Event Composition Model. Sections 4 and 5 explain the EventReactor language and its runtime behavior, respectively. Section 6 illustrates the expressiveness of event modules by means of an example. Section 7 discusses the runtime overhead of EventReactor, and Sections 8 and 9 outline the discussion and future work, respectively.

2 Problem Statement

While RV techniques can be implemented in a GPL and can manually be applied to the base software during the software development process, there is an increasing interest to have DSLs [3–14] for this matter so that the implementation of RV concerns become more abstract and declarative, and the implementation effort is reduced. A closer look at the existing RV DSLs lets us observe three requirements that are typically considered important in the design of these DSLs: a) **abstractness**, b) **modularity**, and c) **compose-ability** of implementations.

The abstractness requirement indicates that suitable domain-specific constructs are needed to implement various kinds of concerns that appear in RV techniques in a declarative, concise, and abstract manner; this is in fact one of the main goals of adopting DSLs instead of GPLs.

In the literature [17], a module is defined as a reusable software unit with well-defined interfaces, which encapsulates its implementation. The modularity requirement indicates that a language must facilitate representing individual concerns that appear in an RV technique as individual modules with well-defined interfaces. The interfaces express the information that the modules provide and require from the other modules for the purpose of runtime verification. The internal implementation of these modules, which is expressed in a DSL, must be encapsulated. If a language falls short to provide a one-to-one mapping between a concern of interest and the modules of a program, the implementation of the concern *scatters across* and *tangles with* the implementation of other concerns in the program [18]. Scattering and tangling are well-known problems in the aspect-oriented community [2], which reduce the modularity and increase the complexity of programs.

The compose-ability requirement means that a language must offer suitable mechanisms to compose individually modularized RV concerns with each other so that the target RV technique is achieved. The example composition mechanisms are explicit method invocation, implicit invocation by means of events, and inheritance. The composition may be constrained, and the constraints must also be modularized and programmed in their DSLs. There are plenty of legacy software systems whose functionality must be extended with RV. Thus modularity and compose-ability must also be considered from the perspective of separating the implementation of RV concerns from the base software and composing these two into an executable system.

To be able to identify the degree to which the aforementioned requirements are fulfilled by the current RV DSLs, in this section we first identify the typical concerns that appear in RV techniques, derive a design space for RV DSLs, and accordingly, we identify the shortcomings of a representative set of RV DSLs. Finally, we illustrate the shortcomings by means of an example.

2.1 Typical Concerns in RV Techniques

Our comprehensive study on the current RV techniques and DSLs [3–14, 19, 20], reveals that four kinds of concerns typically appear in RV techniques: *Base Software*, *Observation*, *Verification*, and *Action*. Figure 1 represents the interactions among these concerns.

- *Base Software* is the software whose correctness must be ensured by an RV technique.
- *Observation* is the concern that abstracts the necessary information for the purpose of runtime verification from the base software. This information can be, for example, the invocation of a method, the value of a variable that is updated, etc.
- *Verification* is the concern that checks the expected and/or unexpected properties of the base software; for this matter, it receives the necessary information from the base software. The verification of the specified properties results in new information, for example, indicating whether the properties are satisfied or violated.

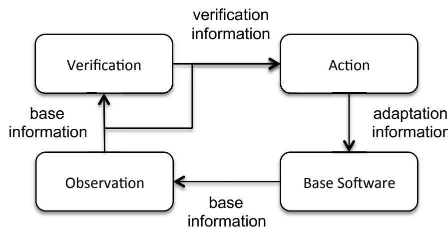


Fig. 1. Typical concerns in RV techniques

- *Action* represents what needs to be performed as the result of verification. Typical examples are diagnosing the causes of failure and recovering the base software from failures [19, 20]. Actions are triggered by the verification concerns, and may exchange information with the base software, for example, for the purpose of recovery.

As Figure 1 shows, in general, RV concerns have a *crosscutting* [2] nature. For example, the observation concerns may crosscut the base software to receive the information from various places in the base software. Likewise, actions may

crosscut the base software, for example, if they need to apply modifications to multiple places in the base software.

Not only the execution of the base software be verified, but also the execution of the verification concerns and actions. For this purpose, the notion of base software can be extended to include verification concerns and actions; higher-level verification concerns can, then, observe and manipulate the execution of this extended base software. Such hierarchal organizations are quite common in adaptive control systems, for example, where multiple levels of control systems can be stacked on each other.

2.2 A Design Space for RV DSLs

The degree to which the abstractness, modularity, and compose-ability requirements are fulfilled by the current RV DSLs differs per language. To establish a basis for the comparison of these languages, in Figure 2, we represent a set of possible alternatives in designing a language for implementing an *RV Concern*.

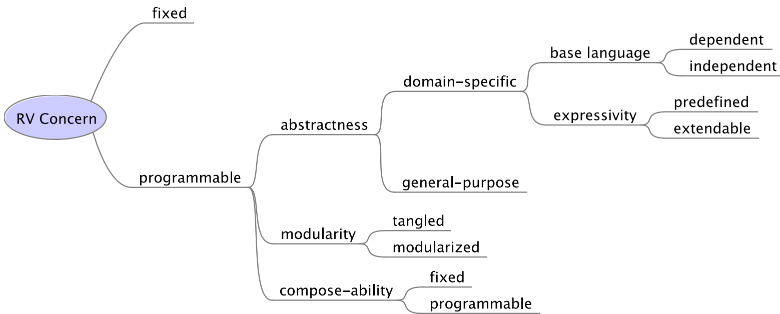


Fig. 2. Design Space of RV DSLs

In Figure 2, the dimension *fixed* means that the implementation of an *RV Concern* is fixed in an RV DSL, or it is not possible at all to implement the concern by the available constructs of the RV DSL; the opposite is *programmable*. If an *RV Concern* is programmable, there are two possibilities in its *abstractness*: *domain-specific* and *general-purpose*. The former means that there are dedicated domain-specific constructs to implement the *RV Concern*, and the latter means that the RV DSL adopts the elements of a GPL for this matter, which naturally reduces the abstractness of implementations. If there are domain-specific constructs, they can either be *dependent* on or *independent* of the language in which the base software is implemented. RV DSLs usually have limited expression power; this is mainly because they provide constructs that are dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The dimension *expressivity* indicates that

the set of offered domain-specific constructs by an RV DSL can be *predefined* or it can be *extendable* with new constructs. Supporting a predefined set of constructs may limit the possibility to program various different kinds of RV concerns in an RV DSL.

As the dimension *modularity* shows, a programmable RV concern may be *modularized* from other RV concerns and the base software; the opposite is *tangled*, meaning that the implementation of a concern is not well separated from the implementation of other concerns. The dimension *compose-ability* indicates that an RV DSL must offer suitable composition operators so that individual RV concerns can be composed with each other to form the target RV technique. An RV DSL can either *fix* the available composition operators, or it can make them *programmable* by offering suitable linguistic constructs for this matter.

2.3 Shortcomings of the Existing RV DSLs w.r.t the Design Space

In this part, we evaluate a representative set of RV DSLs with respect to the design space that is shown in Figure 2. The evaluation is performed for the linguistic constructs that an RV DSL offers for defining the specification of observation, verification, and action concerns.

Table 1. The design space applied to the specification of observation concern

	Fixed	Programmable								
		Abstractness					Modularity		Compose-ability	
		DS				GP	Tangled	Mod.	Fixed	Prog.
		Language Dep.		Expressivity						
		Dep.	Indep.	Pred.	Prog.					
MaCS		+		+				+		+
PQL		+		+			+		+	
Polymer		+		+				+		+
JavaMOP		+		+			+			+
RMOR		+		+				+		+
Trace-matches		+		+			+			+
E-Chaser			+	+				+	+	
TraceContract						+		+	+	
JASS	+									
APP	+									
Spec#	+									
Temporal Rover	+									

Specification of Observation Concerns: An RV DSL may offer dedicated constructs to specify the information that must be abstracted from the base software. This information can be in form of events and/or data values. As Table 1 shows, MaCS [3], PQL [11], Polymer [12], JavaMOP [4], Tracematches [13], RMOR [5], and E-Chaser [10] offer a predefined set of dedicated linguistic constructs for this matter. Among these languages, only the constructs offered by E-Chaser are independent of the language of base software.

TraceContract [14], is an internal DSL [21] which makes use of the libraries offered by the Scala programming language. In TraceContract, the observable information is defined as events, and new kinds of events can be defined. However, as it is inherent for internal DSLs, the specifications are dependent on the GPL in which the internal DSLs are defined. JASS [6], APP [9], Temporal Rover [7], and Spec# [8] do not offer any construct to define the specification of observation concerns. In these languages, the verification concerns and actions are embedded in the base software, and refer to the variables defined within the base software.

Along the modularity dimension, MaCS facilitates modularizing the specifications through so-called *PEDL* (Primitive Event Definition Language) specifications. In JavaMOP, PQL, and Tracematches, the specification of observation concerns is separated from the base software; however, these specifications are tangled with the verification concerns and actions within one module. Polymer facilitates the modularization of specifications by offering so-called *action* modules, which closely resemble Java classes. In RMOR, the specification of observation concerns can be expressed separately from the specification of verification concerns and actions. E-Chaser facilitates the modularization of specification by means of the so-called *superimposition specifications*. In TraceContract, the specifications can be modularized using the modularization mechanism offered by Scala.

With respect to the compose-ability dimension, MaCS facilitates composing primitive fields and variables via Boolean operators. Tracematches adopts the pointcut language of AspectJ [22] to express the specification of observation concerns, and supports Boolean operators to compose pointcut expressions. Similarly, RMOR offers a pointcut language similar to the one in AspectJ or AspectC [23], and supports Boolean operators to compose pointcut expressions. JavaMOP extends the pointcut language of AspectJ with two new predicates; AspectJ pointcuts can be composed with each other via Boolean operators; the JavaMOP-specific predicates can be composed with each other and with AspectJ ones via the conjunction operator. The other languages do not offer constructs for programming the abstracted information with each other.

Specification of Verification Concerns (Properties): As Table 2 shows, all investigated RV DSLs offer dedicated formalisms to specify the properties to be verified. Among these, JavaMOP, E-Chaser, and TraceContract are programmable with new kinds of formalism. Only MaCS, E-Chaser, RMOR, and Tracematches facilitate specifying the verification concerns independently from

Table 2. The design space applied to the specification of verification concerns

Fixed	Programmable								
	Abstractness					Modularity		Compose-ability	
	DS				GP	Tangled	Mod.	Fixed	Prog.
	Language Dep.		Expressivity						
	Dep.	Indep.	Pred.	Prog.					
MaCS		+	+			+		+	
PQL	+		+			+			+
Polymer	+		+			+			+
JavaMOP		+		+		+		+	
RMOR		+	+				+	+	
Trace-matches		+	+			+		+	
E-Chaser		+		+		+		+	
TraceContract					+	+			+
JASS	+		+			+		+	
APP	+		+			+		+	
Spec#	+		+			+			+
Temporal Rover	+		+			+		+	

the language of base software. JavaMOP is also in this category, except for its raw specifications, which are programmed in the Java language.

MaCS offers a dedicated language, called MEDL (Meta Event Definition Language), for the modular specification of verification concerns; however, the composition of verification concerns with observation concerns and actions is tangled in MEDL specifications. Polymer does not have a clear distinction between the specification of verification concerns and actions; together they are considered as security policy, and are tangled within one module. In PQL, JavaMOP, and Tracematches also the specification of verification concerns is tangled with the specification of observation concerns and actions. RMOR facilitates modularization of specifications. TraceContract and E-Chaser facilitate modularizing the specification of verification concerns from the specification of observation concerns; however, the specification of actions remains tangled with the specification of verification concerns. In JASS, APP, Spec#, and Temporal Rover, the specifications of verification concerns are tangled within the base software.

Along the dimension of compose-ability, PQL facilitates programming the composition of properties by means of so-called *sub-queries*; a complex property is the composition of a set of smaller properties expressed as sub-queries. Polymer treats the specifications of policies as Java objects and offer so-called *policy combinators* to compose multiple policies with each other; new policy combinators can be programmed. In TraceContract, properties can be composed in a hierarchical manner through invoking a dedicated function with a variable

length argument list; properties are provided as arguments to this function. Spec# supports inheritance operator to compose the specifications. The other evaluated languages do not facilitate the composition of verification concerns with each other.

Table 3. The design space applied to the specification of actions

	Fixed	Programmable								
		Abstractness					Modularity		Compose-ability	
		DS				GP	Tangled	Mod.	Fixed	Prog.
		Language Dep.		Expressivity						
		Dep.	Indep.	Pred.	Prog.					
MaCS						+		+		+
PQL		+		+			+			+
Polymer		+		+			+			+
JavaMOP						+	+			+
RMOR						+		+		+
Trace-matches						+	+			+
E-Chaser			+	+			+		+	
TraceContract						+	+			+
JASS						+	+			+
APP						+	+			+
Spec#	+									
Temporal Rover					+	+	+			+

Specification of Actions: As Table 3 shows, MaCS, JavaMOP, RMOR, Tracematches, JASS, APP, and Temporal Rover adopt the constructs of the language in which the base software is implemented to express the actions. PQL and Polymer offer a predefined set of dedicated constructs for this matter, which are dependent on the Java language. E-Chaser supports a method invocation as the action, and this is specified independently from the language of base software. Spec# does not offer dedicated constructs for the specification of actions; it raises an exception if the verification fails. TraceContract by default reports the error trace, but it is also possible to program desired actions in the Scala language.

From the perspective of modularity, in all DSLs except MaCS and RMOR the specifications of actions are tangled with the specification of verification concerns. From the perspective of compose-ability, PQL offers a predefined operator, i.e., the sequential composition of specifications. Polymer facilitates composing multiple actions by means of policy combinators. E-Chaser does not facilitate the composition of actions. In the languages in which actions are expressed in a GPL, actions can be composed with each other using the adopted GPL.

2.4 Illustration of the Shortcomings

Since the existing RV DSLs do not span the full design space, the abstractness, modularity, and compose-ability requirements cannot be fulfilled in the implementation of RV techniques if these languages are adopted. In this section, we illustrate these shortcomings by the example of a document-editing software to which runtime verification must be applied.

The document-editing software has the three core modules *Authentication*, *DocumentManager*, and *Storage*. These provide the functionality to authenticate users, to edit a document, and to save its contents on the file system, respectively. *Authentication* and *DocumentManager* are implemented in Java, and *Storage* is implemented in C. Figure 3 shows a UML sequence diagram that depicts the sequence of causally dependent invocations that handle a save request issued by the user. Here, the user first logs in to the system, by invoking the method *login* on the object *anAuthentication*. Then, eventually, s/he starts a save operation by invoking the method *save* on the object *aDocumentManager* of type *DocumentManager*. Subsequently, the functions *open*, *write*, and finally *close* are invoked on *Storage*. The user eventually *logs out* from the application. For the sake of brevity, we omitted the objects that facilitate inter-language communication. We assume that each user modifies one document at a given time, and that the request to modify the document is handled in one causal thread of execution that spans across *anAuthentication*, *aDocumentManager*, and *Storage*.

Assume that we would like to verify the sequence of invocations depicted in Figure 3, to ensure that a request to save a document by an authenticated user eventually results in storing the document on the file system. There are two kinds of failure. First, the save request is issued by an un-authenticated user. Second, after invocation of *save* by an authenticated user, any of the other invocations does not occur in the specified order before the user logs out. As recovery actions, we would like to first log an error message and then prevent the execution of the method whose invocation violates the specified sequence.

We consider two possibilities for implementing the aforementioned runtime check using the available RV DSLs: (a) using a single RV DSL, and (b) using a different RV DSL for each base implementation language.

If we would like to adopt a single RV DSL, the linguistic constructs of the RV DSL must be sufficiently abstract to express runtime behavior of base programs implemented in various different languages. As Table 1 shows, only E-Chaser facilitates abstracting information from the base software implemented in different languages. However, as we show in [10], E-Chaser cannot preserve the modularity of specifications for the base software implemented in multiple languages. Moreover, it cannot verify the causal-dependency of the invocations that span across modules implemented in different languages.

Alternatively, we would like to use a different RV DSL for each language environment. To implement the running example, we use JavaMOP for the Java part

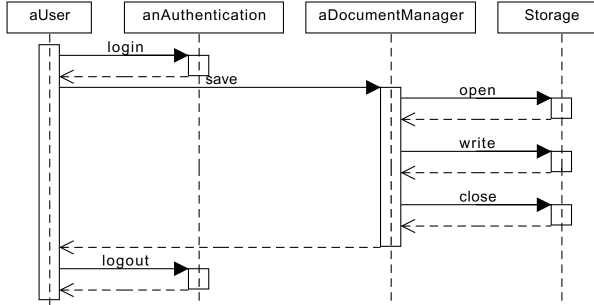


Fig. 3. The sequence of method invocations to save a document

and RMOR for the C part. Listing 1 shows an excerpt of the JavaMOP specification, which defines that the three events `login`, `logout`, and `save` must be abstracted from instances of the Java classes `Authentication` and `DocumentManager`. The events `login` and `logout` represent the state changes after the execution of the methods `login` and `logout` on instances of the class `Authentication`. The event `save` represents the state change before the execution of the method `save` on instances of the class `DocumentManager`. The regular expression in line 8 indicates that somewhere between the occurrence of `login` and `logout` (i.e., only when the user is authenticated), the event `save` may occur. The specification in lines 9 to 12 indicates that an error message must be shown if the verification of the regular expression fails, and the execution of the corresponding method must be prevented.

```
1  event login after(Authentication a) :
2                  execution(* Authentication.login()) && target(a) {}
3  event logout after(Authentication a) :
4                  execution(* Authentication.logout()) && target(a) {}
5  event save before(DocumentManager d) :
6                  execution(* DocumentManager.save()) &&
7                  target(d) {}
8  ere : (login save* logout)*
9  @fail {
10     System.err.println("Problem in saving the document!");
11     __SKIP;
12  }
```

Listing 1. A specification of the Java part

Listing 2 shows an excerpt of the RMOR specification, which defines that the three events `open`, `write`, and `close` must be abstracted from the C module `Storage`. The specification of the state machine in lines 4 to 7 indicates that when the event `open` occurs, a transition to the state `Opening` must take place,

which expects the event `write` to be the next event that occurs. If the event `write` or any other event occurs in the state `Opening`, there will be a transition to the built-in state `error`, which indicates that this is an unexpected event.

```
1 event open = before execution(Storage.c:open);
2 event write = before execution(Storage.c:write);
3 event close = before execution(Storage.c:close);
4 initial state Opening {when open -> Opening; when write->error;/*...*/}
5 live state Opening {when write -> Writing;/*...*/}
6 live state Writing {when close -> Closing;/*...*/}
7 live state Closing {/*...*/}
```

Listing 2. A specification of the C part

With the state-of-the-art RV DSLs, the specifications are lacking abstractness because they are dedicated to one GPL. This prevents us from specifying the desired properties of the base software in a correct way. For example, we need to specify that the sequence of events specified in lines 4 to 7 of Listing 2 must occur after the event `save` specified in Listing 1. However, this can neither be specified in JavaMOP nor in RMOR; consequently, we have to provide a third DSL dedicated for expressing the compositions of these specifications, which can be a costly task. It is therefore preferable that an RV DSL offers linguistic constructs that are sufficiently abstract to deal with software implemented in various languages.

Since the adopted specification languages make use of the elements of a GPL, we were obliged to sacrifice the modularity of specifications by splitting them based on the implementation language of base software; in our example, the specification of sequence of events is divided in two modules. As a consequence, compose-ability of specifications is reduced since there is no standard linguistic mechanism for composing the specifications that are expressed in various RV DSLs.

3 Towards an RV Language Composition Framework

The shortcomings of the current RV DSLs in expressing, modularizing, and composing diverse kinds of RV techniques may consequently oblige software engineers to design and implement new RV DSLs. However, the design and implementation of a new RV DSL from scratch requires extensive knowledge of language design, and may be a time-consuming task. Moreover, the existing RV DSLs share several abstractions, unfortunately, without sharing an implementation.

To ease the design and implementation of RV DSLs, we advocate the need for a language composition framework, which offers the necessary linguistic mechanisms to define new DSLs while providing the mechanisms to achieve modularity and compose-ability in the implementation of domain-specific crosscutting concerns. To this aim, this section first identifies the characteristic features of RV concerns that must be respected by such a framework. Afterwards, it explains

Event Composition Model that is a base model for such a framework, and discusses its suitability for this matter.

3.1 Characteristic Features of RV Concerns

A closer look at the model depicted in Figure 1 and the languages discussed in the previous section reveals the following characteristic features of RV concerns.

First, the interactions among the concepts of RV techniques have *by nature a transient characteristic*, meaning that the changes in the states of a concern drive the other concerns. For example, the verification concerns observe the changes that occur in the states of the base software, and verify the state changes of interest against the specified properties of the base software. Various RV techniques may require to consider various kinds of state changes; examples are a time-out value, and an invocation of a method on an object. This implies that an RV language composition framework must facilitate defining *open-ended kinds of state changes in the base software as well as in RV concerns.*

Second, it is not easy or even possible to foresee all kinds of concerns that appear in the RV techniques of today or in future. For example, some techniques require to specify and verify the sequence of method invocations in the base software, whereas some others may require to specify and verify the properties of operating system processes. The steady development of new RV DSLs that support new kinds of formalisms to specify the properties is a consequence of this. Therefore, as the second requirement, we claim that an RV language composition framework must facilitate implementing *open-ended kinds of RV concerns* such that the specifications are modular.

Third, although Figure 1 shows a fixed hierarchy of concerns, in a general case, the kinds of compositions cannot be fixed. For example, an RV technique itself may be considered as the base software whose behavior must be checked at runtime, a specification may be composed of multiple sub-specifications, etc. This indicates that an RV language composition framework must facilitate implementing *open-ended kinds of compositions.*

Finally, due to the increasing number of multi-language software systems (e.g., embedded software) to which RV techniques must be applied, an RV language composition framework must support *open-ended sets of base languages.*

3.2 Event Composition Model

In [15, 16], we introduced Event Composition Model as a model, which respects the aforementioned characteristic features. In this section, we explain a revised version of this model whose concepts are shown in Figure 4 via a UML class diagram.

At a high level of abstraction, Event Composition Model considers the execution *Environment* as a set of *Events* and *Event Modules.* In software systems, events typically represent changes in the states of interest and are means for abstracting the execution trace of programs. As the class *Event Type* shows,

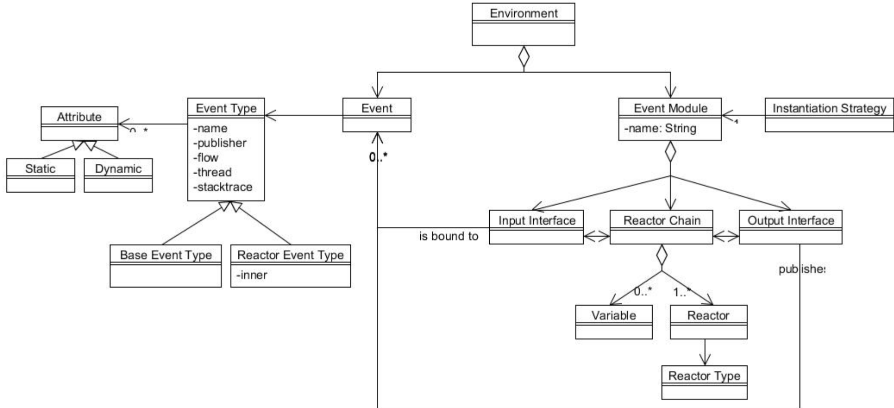


Fig. 4. Event Composition Model

events are typed entities; *Base Event Type* and *Reactor Event Type* are two main specializations. The former represents the events that occur in the base software, and the latter represents the events that are published by so-called event modules that will be explained in the subsequent paragraphs. Event Composition Model does not fix the event types and events; new kinds of application and/or domain-specific event types and events can be defined in the language composition framework.

As the class *Attribute* in Figure 4 shows, each event type defines a set of attributes for events; these are means to keep the abstracted information from the execution trace of software. The attributes are classified into *Static* and *Dynamic*. The former includes the set of attributes whose values do not change and are known at the time an event is defined in the framework. The latter defines the set of attributes whose values are known when an event is published during the execution of software. For example, for an event that corresponds to the invocation of a method, the name and the value of parameters can be defined as static and dynamic attributes of the event, respectively.

Event Composition Model considers *name*, *publisher*, *returnflow*, *thread*, and *stacktrace* as predefined attributes. These attributes respectively specify the unique name of an event in the framework, the publisher of the event, and the changes that must be applied to the flow of execution of the publisher after an event is successfully processed, the thread of execution in which the event is published, and a report of the active stack frames at the time the event is published. For the reactor events, the attribute *inner* keeps a reference to the input event being processed by an event module. More application and/or domain-specific attributes can be defined for each type of event.

Event Composition Model introduces *Event Module* as a means to modularize a group of related events and the reactions to them. In software engineering, a module is usually considered as a referable entity with well-defined interfaces.

Two kinds of interfaces, known as input and output, are typically considered for a module. The former defines the services that a module requires from its context; the latter specifies the services that a module provides to its context. A module also has an implementation part, which is bound to its interfaces to provide the specified interfaces. Modules promote encapsulation by utilizing interfaces as their interaction points with their context.

As Figure 4 shows, event modules adhere to the above definition of modules in the following ways. An event module is identifiable and referable by its unique *name*. An event module has an *Input Interface*, an implementation – which is termed as *Reactor Chain*–, and an *Output Interface*; these elements are bound to each other.

The input interface of an event module specifies the set of events of interest to which the event module must react. Event Composition Model does not fix the semantics for selecting the events of interest and for binding them to the input interface of an event module. One important difference between the input interface of modules in programming languages and the input interface of event modules is that in programming languages input interfaces are invoked explicitly, whereas in event modules invocations are implicit. The explicit invocation means that programmers write code for invoking the input interface of a module. In contrast, implicit invocation [24] means that there is no need for such code, and when an event of interest occurs, the corresponding event module is activated by the language composition framework.

Figure 4 shows that the implementation of an event module contains a set of *Reactor*s and *Variable*s. Each reactor processes (a subset of) the events specified in the input interface of the event module. Reactors are typed entities; a *Reactor Type* is a domain-specific type that defines the semantics in processing the events of interest.

Reactors can be composed with each other within a reactor chain. Such reactors can exchange information among each other via the variables defined within the corresponding reactor chain. While processing an input event, a reactor may publish new events, which are termed as reactor events. Via the attribute *inner* a reference to the input event being processed by the reactor is maintained.

The output interface of an event module defines the set of events that are published by the event module to its context. To be able to process events, event modules are instantiated during the execution of software based on some *Instantiation Strategy*; the strategies can be programmed according to the application/domain demands.

3.3 Motivations to Adopt Event Composition Model

Event Composition Model respects the characteristic features of RV concerns identified in Section 3.1 in the following ways. Events are means to implement the *transient nature* of RV concerns. Various kinds of RV concerns can uniformly be implemented and modularized as event modules. The support for domain-specific reactor types facilitates expressing the RV concerns of interest

at a higher abstraction level in their DSL. *Open-ended kinds of RV concerns* can be programmed because Event Composition Model is open-ended with respect to event types, events, reactors, reactor types, event modules, and instantiation strategies.

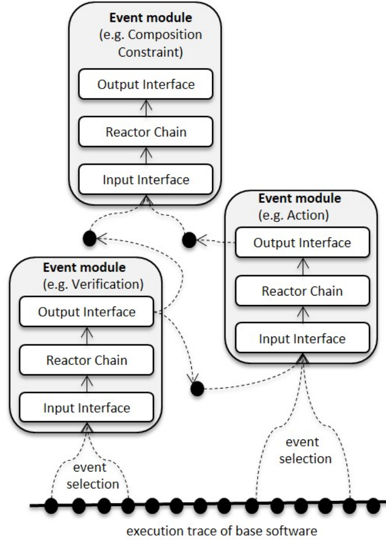


Fig. 5. Modularizing RV concerns via event modules

The events in the output interface of an event module can further be specified as the input interface of other event modules, this facilitates composing event modules with each other and defining the composition constraints via event modules. *Open-ended kinds of compositions* can be programmed according to the application and/or domain requirements. Event modules only interface with the base software in terms of events. Regardless of the implementation language, events are a universal principle in the execution of all software systems; and Event Composition Model does not constrain the kinds of events to be supported. Therefore, Event Composition Model is *open-ended with respect to the supported base languages.*

There is an analogy between the concepts of Event Composition Model and the ones in the aspect-oriented languages. Here, events correspond to join points, and event modules correspond to aspects. Since multiple events can be selected via the input interface of an event module, and such events may be published by various publishers, event modules can be adopted to modularize crosscutting concerns. The support for domain-specific reactor types facilitate expressing such concerns in their DSLs.

Figure 5 shows an example way of implementing RV concerns via event modules. Here, the input interface of the event module *verification* selects the events

of interest that are published by the base software. These events are provided to the reactor chain, which implements the functionality of the verification. If the verification fails, the result is published as a reactor event, which is bound to the output interface of *verification*. This event is received by the input interface of the event module *action*, which depending on the kinds of action may collect more events from the base software. Composition constraints among the modules can further be defined and modularized via event modules.

4 The EventReactor Language Composition Framework

EventReactor is a language composition framework that provides dedicated linguistic constructs to implement the concepts of Event Composition Model. In [15, 16], we explain an earlier version of EventReactor by means of an example RV technique. This paper proposes a new version of EventReactor that offers linguistic constructs to modularly define event types and events. In addition, it facilitates the explicit definition of the output interface of event modules so that the boundary of event modules are more explicit in the specifications. The new version of EventReactor supports programmable instantiation strategies of event modules based on event attributes; this gives flexibility to programmers to specify the desired strategies. In the following, we make use of our illustrative example to explain EventReactor language. For the sake of brevity, some details are eliminated; the full implementation of the example can be downloaded from the website of EventReactor[1].

4.1 Specification of Event Types and Events

Event types are data structures that define a set of static and dynamic attributes for events. EventReactor provides four built-in event types `EventType`, `BaseEventType`, `ReactorEventType`, and `MethodBased`; it also offers a dedicated language to define custom event types modularly.

Complying with Figure 4, the built-in event type `EventType` is the super type of all other event types, and `BaseEventType`, `ReactorEventType` are two specializations that define the specified attributes in Figure 4. For the attribute `returnflow`, EventReactor currently supports the following values: `Continue` means that the flow of execution must not be changed; `Exit` means that the execution of program must terminate; `Return` means that the flow of execution must return to the publisher.

The event type `MethodBased` represents the state changes corresponding to the invocation and execution of methods in the base program, as it is supported by the current aspect-oriented languages [22]. Listing 3 shows the definition of the event type `MethodBased`, which extends the type `BaseEventType` (Line 1) and thereby inherits all its attributes. In addition to the inherited attributes, this type declares further static context attributes (Lines 3–5) and dynamic context

[1] `http://sourceforge.net/projects/eventreactor/`

attributes (Lines 7–8). The static attributes define the kind of event (i.e., before invocation, before execution, after invocation, after execution), the signature of the method, and the module in which the method is defined. The dynamic attributes define the arguments of the method and the target object in which the method is executed.

```
1 eventtype MethodBased extends BaseEventType{
2   staticcontext:
3     kind : MethodBasedEvents;
4     signature : String;
5     module : String;
6   dynamiccontext:
7     args : Object [];
8     target : Object;
9 }
```

Listing 3. The specification of MethodBased

EventReactor offers a dedicated language to programmers to define the events of interest modularly. For this matter, programmers must specify its unique name, its event type, and the values of the static attributes defined by the event type. For the built-in method-based events, the compiler of EventReactor extracts the relevant information from the base software, and automatically creates event definitions in this language. The compiler adopts the same approach as the Compose* compiler [25] to support multiple base language (e.g., as Java, C, and .Net languages) that adhere to the abstract language model provided by Compose*.

Listing 4 shows an example of an event definition that is generated by the EventReactor compiler. The event is `MethodBased` (Line 1) and represents the state change after invocation and immediately before the execution (Line 3) of the method `save` (Line 4) defined in the class `DocumentManagement` (Line 5).

```
1 event saveDocumentManager instanceof MethodBased{
2  staticcontext:
3    kind = MethodBasedEvents.BeforeExecution;
4    signature = "public void save(java.lang.String, java.lang.Object)"
5    module = "public class DocumentManagement extends java.lang.Object"
6 }
```

Listing 4. The specification of a method-based event

To publish an event, it is necessary to initialize its dynamic attributes and inform the runtime environment of EventReactor of the event. The API of EventReactor offers two routines for this matter. In the first one, the information about the event is provided as a comma-separated list of attributes and their values. This API is useful if the base software is implemented in a language other than Java. The second API is useful if the events are published from a Java program.

In this case, EventReactor generates Java classes from the specification of events, whose instances represent events.

Listing 5 shows an example code for publishing the `saveDocumentManager` event. First, an instance of the generated class `saveDocumentManager` is constructed (Line 1). Next, the dynamic context attributes are set (Lines 2–3). And finally, the event object is published to the EventReactor runtime (Line 4).

```
1 saveDocumentManager event = new saveDocumentManager();
2 event.initializeDynamicAttribute("publisher", aDocumentManager);
3 // initialize other dynamic attributes, e.g., the current thread
4 EventReactor.publish(event);
```

Listing 5. Publishing a method-based event

For the method-based events the EventReactor compiler instruments the base program to publish the defined events. To publish EventReactor events from base software written in a language other than Java, Java-JNI[2] is used to access the runtime environment of EventReactor.

4.2 Specification of RV Concerns

EventReactor offers `eventpackage` as a means to package the specification of RV concerns. The elements defined within an event package can be referred to by their fully qualified name. Like any other programming language, programmers decide how the implementations must be packaged.

```
1  eventpackage ObservationConcern{
2  selectors
3     save = {E | isBeforeExecution(E, M),
4                  isMethodWithName(M,'save'),
5                  isModuleWithName(C,'DocumentManager'),
6                  isDefinedIn(M, C)};
7     open = ...
8     write = ...
9     close = ...
10    login = ...
11    logout = ...
12    failure = {E | isEventWithName(E, 'violated'),
13                  isEventModuleWithName(EM, '*.Verification'),
14                  isPublishedBy(E, EM)};
15  }
```

Listing 6. Specification of observation concern

[2] See homepage of Java-JNI: `http://download.oracle.com/javase/1.5.0/docs/guide/jni/spec/jniTOC.html`

As Listing 6 shows, the events of interest are selected by means of queries in the Prolog language. In lines 3–6, the Prolog expression specifies that the method-based events `E`, which correspond to the state change after the invocation and immediately before the execution of the method `''save''` defined in the class `''DocumentManager''`, must be selected. The other base events that must be verified are selected likewise in lines 7–11. Lines 12–14 specify that the event `violated`, which is published by the event module `Verification`, is another event of interest to be selected.

A modular implementation of the verification concern is provided in Listing 7 via the event module `Verification`. As Figure 4 shows, an event module must have an input interface, which must be bound to a set of events of interest defined in the framework. Lines 3–5 define the input interface, which is bound to the named selectors defined in the event package `ObservationConcern`.

In EventReactor, the instantiation strategy can be specified as a comma-separated list of event attributes. Such a specification indicates that a distinct instance of the event module must be created for each distinct combination of the values of the specified attributes. If no instantiation strategy is specified, the event module will be instantiated in a singleton manner. Line 6 of Listing 7 indicates that the event module must be instantiated per thread.

Line 7 binds the reactor chain `Verify` to the input interface, and defines the property to be verified as a regular expression over the input events, which is passed to the `Verify` reactor chain. The event `violated` is specified as the output interface of the event module in line 8; this event is published from within the reactor chain if any of the input events does not occur in the specified order.

```
1 eventpackage VerificationConcern{
2  eventmodules
3    Verification := {ObservationConcern.login, ObservationConcern.logout,
4                     ObservationConcern.save, ObservationConcern.open,
5                     ObservationConcern.write, ObservationConcern.close}
6                   {'thread'}
7                     <- Verify('(login (save open write+ close)* logout)*')
8                     -> {violated : ReactorEventType};
9   }
```

Listing 7. An event module for the verification concern

In EventReactor, reactor chains are defined separately from event module to facilitate reusing them in multiple event modules. Reuse is further increased by the possibility to parameterize reactor chains. An example of a parameterized reactor chain is given in Listing 8, which shows the reactor chain `Verify` used in the event module discussed above. The chain consists of a single reactor named as `regexp` of the type `RegularExpression`. As for reactor chains, reactors can be parametric. This is shown in line 3, in which the parameter `regformula` is assigned to the reactor's parameter `expression`.

```
1 reactorchain Verify(regformula: String){
2  reactors
3    regexp: RegularExpression = { reactor.expression = regformula; };
4 }
```

Listing 8. A reactor chain for the verification concern

In addition to the verification concerns, recovery actions can also be defined via event modules. Listing 9 defines the event package `RecoveryConcern` in which the event module `Recovery` is defined. Here, the event `failure`, which is published by the event module `Verification` and selected in Listing 6, is specified as the input interface of the event module `Recovery`. The instantiation strategy `"publisher"` indicates that individual instances of the event module must be created for individual instances of the event module `Verification` that publishes the events `violated`. The reactor chain `Recover` as the implementation of the event module; the event module does not publish any event as its output interface.

```
1 eventpackage RecoveryConcern{
2  eventmodules
3    Recovery := {ObservationConcern.failure} {"publisher"} <− Recover() −> { };
4 }
```

Listing 9. An event module for the recovery concern

Listing 10 defines the reactor chain `Recover` in which two reactors `logger` and `preventer` are defined of the types `Log` and `ForceReturn`, respectively. Since the reactor `logger` is first in the chain, it processes the input event first. Afterwards the event is processed by `preventer`, which suppresses the further execution of the base method whose execution has caused the failure. The runtime behavior of running example in processing events is explained in Section 5.

```
1 reactorchain Recover(){
2  reactors
3    logger: Log = {reactor.message = 'An error has occurred!';};
4    preventer: ForceReturn;
5 }
```

Listing 10. A reactor chain for the recovery concern

4.3 Implementation of Reactor Types

Reactor types are means to encapsulate the functionality for processing input events and publishing output events. Each reactor type is defined via a so-called reaction class and a specification of meta information. The reaction class, which is implemented in Java, provides the functionality of reactor type in processing input events. The specification of meta information defines the name of the

reactor type, the name of its reaction class, the name and type of reactor events that are published by the reactor type, and the parameters of the reactor type. Listing 11 shows the specification of the reactor type `RegularExpression`.

Listing 12 shows the implementation of the class `RegExpClass`. Each action class must extend the class `ReactorAction` that is provided by EventReactor, and must implement two methods `initialize` and `execute`. The former is executed when the corresponding reactor is instantiated, the latter is executed when the corresponding reactor receives an event to process. These methods can access the instances of the corresponding reactor, reactor chain, and event module via their argument `context`.

```
1 reactortype RegularExpression {
2     reaction = RegExpClass;
3     events = {violated : ReactorEventType};
4     parameters = {expression : String};
5 }
```

Listing 11. The specification of the RegularExpression reactor type

```
1  public class RegExpClass extends ReactorAction {
2   Automaton automaton;
3   @Override
4   public void initialize(Context context) throws Exception {
5     String expression = this.getParameters("expression");
6     automaton = //parse the regular expression and create the automaton
7     // ...
8   }
9   @Override
10  public void execute(Event event, Context context) throws Exception {
11   boolean failed = automaton.check(event);
12   if (failed == true) {
13        violated result = new violated();
14        result.initializeDynamicAttribute("inner", event);
15        result.initializeDynamicAttribute("publisher", context.eventmodule);
16        // ...
17        EventReactor.publish(violated);
18   }
19  }
20 }
```

Listing 12. The action class for the reactor type RegularExpression

In Listing 12, the method `initialize` parses the regular expression provided via the parameter `expression`, and it creates an automaton according to the algorithm presented in [26]. The method `execute` makes use of the generated automaton to verify `event` against the regular expression formula, and concludes the success or failure of the verification. As lines 13–17 show, if the verification fails, it creates a reactor event and publishes it. Here, the attribute `inner` is

initialized with `event` whose processing causes the reactor event be published. The corresponding event module is specified as the publisher, and other dynamic attributes can be initialized likewise.

For the sake of brevity, this paper does not show the implementation of the reactor types `Log` and `ForceReturn`. In short, the reactor type `Log` reports a message on the screen when it receives an event to process. The message is passed to the reactor type as a parameter. The reactor type `ForceReturn` prevents the invocation or the execution of a method to proceed. For this matter, it checks whether the input event is of the type `MethodBased` and has the value `BeforeInvocation` or `BeforeExecution` in its attribute `kind`. If so, it assigns the value `Flow.Return` to the attribute `returnflow` of the input event. As explained in Section 5, the runtime environment of EventReactor changes the flow of execution accordingly. If `ForceReturn` receives a reactor event to process, it traverses the chain of causally dependent events via the attribute `inner` of the reactor event until it reaches to an event of the type `MethodBased`, and performs the aforementioned operation.

5 Runtime Event Processing in EventReactor

As explained in [15, 27], the specifications of event types, events, reactor types, event packages, and reactor chains are input to the EventReactor compiler, which performs various checks on the specifications to ensure their correctness. The compiler can also identify method-based events in a program. For this matter, the program must be provided as input; as the output, the program is instrumented with so-called *notifier* code, which implements the functionality to publish events to the runtime environment of EventReactor, and to get the results back when the event is processed.

In object-oriented programs, objects are regarded as standard publishers. Upon the creation of an object of interest, a *notifier* is bound to the object. In non-object-oriented programs, software files are regarded as standard publishers to which a *notifier* is bound at compile time. At runtime, a *notifier* assigns a unique identifier to the corresponding file to be used as the unique identifier of the publisher for the corresponding method-based events. If an event module publishes events as its output interface, a *notifier* is bound to the event module upon the instantiation of event module. For the programmer-defined events, programmers must implement the functionality to publish the events using the interface provided by EventReactor.

As explained in Section 4, an event is defined with a unique name, a set of static and dynamic attributes. The name and the static attributes of events are fixed, and cannot be changed when an event is published to the runtime environment. As a result, the selectors that query events based on their name and/or static attributes can be evaluated at compile time, and the events that form the results of each selector can be tagged. The EventReactor compiler maintains these tags and the information about the specifications in a *repository*, which is used by the runtime environment to process events. In the following subsection,

we first explain the algorithm adopted by EventReactor to process runtime events. Second, we illustrate the runtime behavior of our running example. We will discuss the runtime overhead of EventReactor in Section 7.

5.1 The Execution Semantics of EventReactor

The runtime environment of EventReactor makes use of Algorithm 1 to receive events and process them in a synchronous way. As line 1 shows, first it is checked whether the event is known in the language; this check is performed by matching the unique name of the event against the unique name of events that are defined in the language. If the event does not match any defined event, the runtime environment ignores it. Otherwise, as line 3 shows, the runtime environment retrieves the set of selectors to which this event matches, using the tags generated by the compiler. If the event matches multiple selectors, the runtime environment considers a random order among the selectors. As line 4 shows, for each selector, the set of event modules that refer to the selector in their input interface is retrieved. If there are multiple event modules referring to the selector, the runtime environment considers a random order for the event modules, unless their order is specified by the programmer using the keyword *precede* [15]. Lines 5 to 12 deal with the instantiation of each event module. To maintain a reference to the instantiated event modules, the runtime environment of EventReactor creates a so-called **event module table** for each specified event module. An event module table resembles a relational database table [28], with two columns *col_reference* and *col_index*. The latter keeps a reference to a distinct instance of the corresponding event module, and the former is the primary key in the table. If the event module is specified to be instantiated as singleton, there will be only one row in this table. Otherwise, there will be one row for each distinct combination of attributes that form the instantiation strategy of the event module. In this case, the distinct combination of attributes that form the instantiation strategy are used for indexing.

Line 5 of Algorithm 1 retrieves the corresponding table of an event module; if there is no such table, it means that it is the first time that the event module is instantiated. In this case, line 7 creates a table for the event module. Line 9 retrieves the corresponding instance of the event module based on the specified instantiation strategy for the event module. If there is no instance, line 11 creates one and inserts it in the event module table. In line 13 to 28, the event is provided to the corresponding instance of the event module to be processed. For each specified reactor in the implementation of the event module, line 14 checks whether the event is of interest; as it is shown in Listing 18 in page 58, each reactor can specify the set of events to which it reacts. If the event is of interest, the reactor starts executing.

In lines 15 to 20 show that while executing, a reactor may publish reactor events; this causes the execution of the reactor to be suspended until the reactor event is processed. Line 17 checks whether a reactor event matches any of the events specified in the output interface of the corresponding event module. If so,

Algorithm 1. Process(event, repository)

```
1  e ← Match(event, repository.DefinedEvents);
2  if e ≠ null then
3      foreach selector in e.MatchedSelectors do
4          foreach eventmodule in selector.MatchedEventModules do
5              table ← repository.GetTable(eventmodule);
6              if table = null then
7                  table ← repository.CreateTable(eventmodule);
8              end
9              instance ←table.GetInstance(eventmodule);
10             if instance = null then
11                 instance ←table.CreateRow(eventmodule);
12             end
13             foreach reactor in instance.ReactorChain do
14                 if reactor.Match(e) then
15                     while reactor.Action not terminated do
16                         reactorevent ← reactor.Action.Execute(e);
17                         if Match(instance.OutputInterface, reactorevent) then
18                             Publish (reactorevent);
19                         end
20                     end
21                 end
22                 if e.returnflow = Exit then
23                     Terminate Program;
24                 end
25                 if e.returnflow = Return then
26                     Return To Publisher;
27                 end
28             end
29         end
30     end
31 end
```

in line 18 the *notifier* bound to the event module publishes the event to the runtime environment of EventReactor. Consequently, the event becomes available to be processed with the same algorithm.

While processing an input event, among other dynamic attributes, a reactor may change the value of the attribute *returnflow*. If the execution of the reactor terminates successfully, the runtime environment checks the value of this attribute, and changes the flow of execution accordingly. If the value is *Exit*, the execution of the program terminates. If the value is *Return*, it means that the event processing must not proceed with the other reactors or event modules, and the flow of execution must return to the publisher. For the method-based events, when the flow of execution returns to the publisher, the *notifier* bound to the publisher prevents the invocation and/or the execution of a method to proceed if the value *return* is specified for *returnflow*.

5.2 An Illustration of the Runtime Behavior

Figure 6 shows the runtime view of our illustrative example. Here, *anAuthentication*, *aDocumentManager*, and *Storage* are the application modules of interest to which the notifiers *Notifier(A)*, *Notifier(D)*, and *Notifier(S)* are bound. If any of the events of interest occurs during the execution of the base program, instances of the event modules *Verification* and *Recovery* are created and managed by *Runtime Environment*. Since the event module *Verification* publishes an event as its output interface, the notifier *Notifier(V)* is bound to it so that the event can be provided to the *Runtime Environment*.

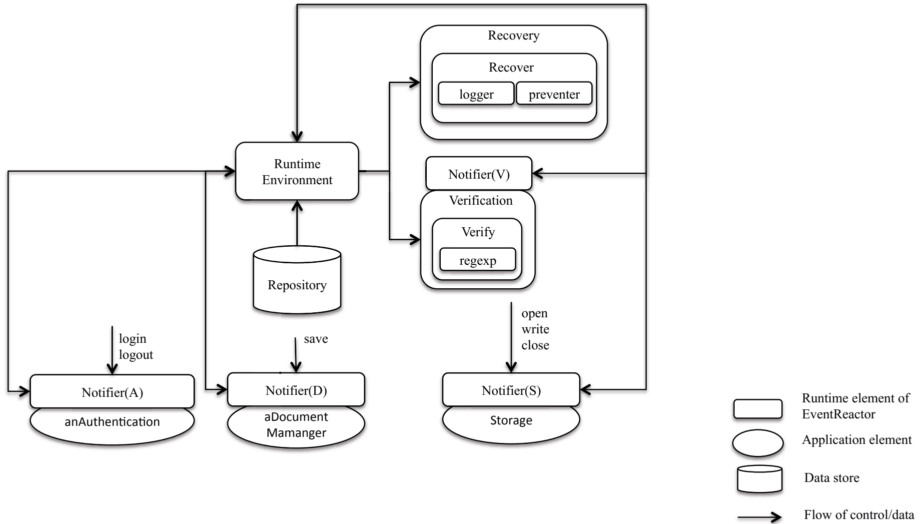


Fig. 6. A runtime view of the illustrative example

In runtime verification, we always need to specify when a verification must start and when it must terminate. In our example, we assumed that the events *login* and *logout* always occur and perform as the start and end point of the verification. Assume for example that in the causal thread of execution t a user invokes the method *login* on the object *anAuthentication*. Before the execution of this method, *Notifier(A)* informs the *Runtime Environment* of an event representing the corresponding state change. According to the specification in Listing 7, this event is of interest and is part of the input interface of the event module *Verification*.

Since there is no event module table named *VerificationConcern.Verification*, the *Runtime Environment* creates one. Afterwards, it inserts one row in this table, inserts the value t in the column *col_index*, creates a new instance of the event module to which *Notifier(V)* is bound, and inserts a reference to the

created instance in the column *col_reference*. Suppose that the user wishes to save a document by invoking the method *save* on *aDocumentManager*, which consequently *Notifier(D)* publishes the corresponding event to the *Runtime Environment*. There is already an event module table named as *VerificationConcern.Verification*, which means it is not the first time that an event of interest for the event module *Verification* is processed. Here, the *Runtime Environment* retrieves the row whose *col_index* matches t; there is one row, and the event is provided to the corresponding instance of the event module to be processed.

Assume that instead of the function *open*, the function *write* is invoked on *Storage*, and the corresponding event is published to the *Runtime Environment* by *Notifier(S)*. Similar to the previous case, the event is provided to the corresponding instance of the event module to be processed. Since the event violates the expected sequence of events, the reactor *regexp* publishes the reactor event *violated*, which is be bound to the event *violated* specified in the output interface of the event module *Verification*, and is published to the *Runtime Environment* by *Notifier(V)*. Consequently, the *Runtime Environment* creates an event module table named as *RecoveryConcern.Recovery*. The *Runtime Environment* inserts the unique identifier of the corresponding instance of the event module *Verification* in the column *col_index*, creates an instance of the event module *Recovery*, and provides the event to it. The instance of the reactor *logger* receives the event, and reports the specified error message on the screen, afterwards the instance of the reactor *preventer* prevents the execution of the method *write* on *Storage*.

If another user initiates a save operation in another causal thread of execution, a new sequence of causally dependent events must be verified. Therefore, a new row is created in the table *VerificationConcern.Verification* and a new instance of the event module *Verification* is created for this matter.

6 Illustration of the Expressiveness of Event Modules

As we explained in Section 3.1, it is generally not possible to fix the kinds of base concerns whose properties must be expressed, and the kinds of RV concerns. This amplifies the need for supporting open-ended kinds of RV concerns by an RV DSL. Otherwise, programmers have to provide workaround implementations using the available constructs in an RV DSL, and as we illustrate in [15, 16], such workarounds tend to be very complex, barely modular, little compose-able and at lower levels of abstraction. The current RV DSLs, however, do not support open-ended kinds of RV concerns. These languages typically support objects or source files as the publisher of events, and only support verifying the temporal properties of a group of events [4, 13, 11].

To illustrate the need for supporting open-ended kinds of RV concerns, this section makes use of *Recoverable Process* [20] as an example RV technique. Recoverable Process aims at making processes fault-tolerant by monitoring processes to detect their failures, and by either restarting a single failed process (called the

local recovery strategy) or a group of semantically related processes, including the failed one (called **global recovery** strategy).

In [15], we provide an example implementation of Recoverable Process in an existing RV DSL, and explain in details the problems regarding the modularity, compose-ability, and abstractness of implementations, which arise due to the lack of support for open-ended kinds of RV concerns. In [27] we also explained the shortcomings of current GPLs in implementing Recoverable Process in a modular way. This section explains a possible implementation of Recoverable Process in EventReactor, and discusses the suitability of the constructs offered by EventReactor to preserve the modularity, compose-ability, and abstractness of implementations.

6.1 Recoverable Process by Example

Figure 7 is a UML class diagram representing the concerns in Recoverable Process. *AppProcess* represents a child process, and has the attributes *pid*, *name*, *status*, *init*, and *kill*. The attribute *pid* is the unique identifier of the child process, which is generated by the operating system. The attribute *name* is the developer-specified name of the child process. The attribute *status* is the execution state of the child process, which can either be *running*, *terminated*, or *under-recovery*. The attributes *init* and *kill* are the methods that create or kill the child process, respectively. The events *initiated* and *killed*, which are shown as operations in the figure, occur if the child process is created or killed, respectively.

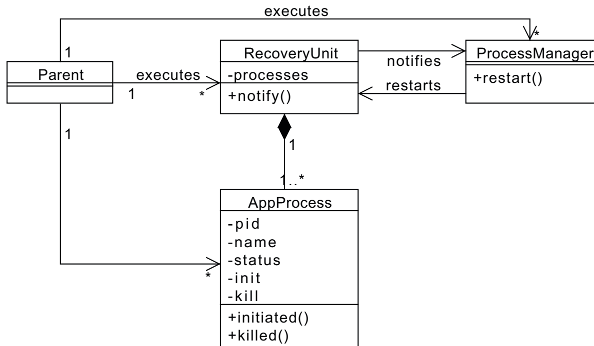


Fig. 7. The concerns of Recoverable Process

The concern *RecoveryUnit* represents a group of child processes that must be recovered together. *RecoveryUnit* detects the failures in the corresponding child processes, and publishes an event to the concern *ProcessManager* to inform of the failures. Consequently, *ProcessManager* recovers the corresponding child

processes by changing their status to *under-recovery*, restarting them, and setting their status back to *running*. The concern *Parent* represents the parent process of *AppProcess*. It creates the child processes, and publishes the event *initiated* for each of them.

Assume for example that we would like to apply Recoverable Process to an example media-player software to make its processes fault-tolerant. An abstract block diagram of the media-player software is shown in Figure 8. The software is structured around the five processes *Runner*, *UserInterface*, *MPCore*, *Audio* and *Video*, which execute the modules *Main*, *GUI*, *Core*, *Libao*, and *Libvo*, respectively. The nesting of blocks shows that the parent process *Runner* has spawned the other processes as children. The arrows in the figure represent the messages that are exchanged among processes. With a global recovery strategy, the child processes *Core*, *Audio*, *Video*, *User Interface* can be restarted as a group. When local recovery is applied, the processes can be restarted individually.

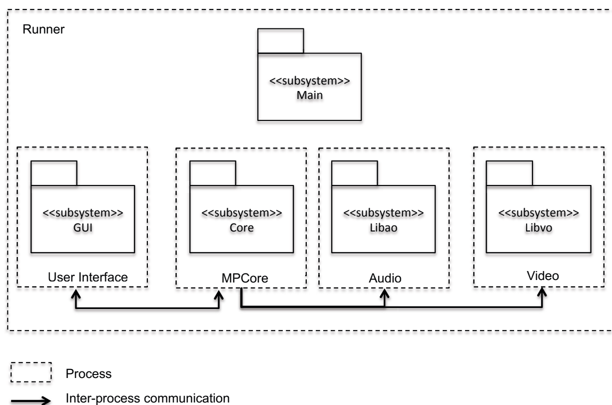


Fig. 8. An abstract block diagram of the media-player software

6.2 Recoverable Process in EventReactor

Defining and Publishing Events: As the concern *AppProcess* in Figure 7 shows, two events *initiated* and *killed* must be defined for each child process of interest. Listing 13 shows the specification of the event type *ChildProcessEvent*. Here, the attributes `PID` and `parent` represent the unique identifier of the child process and its parent. The events `MPCoreInitiated` and `MPCoreKilled` are defined of the type `ChildProcessEvent` to represent the initialization and destruction of the child process *MPCore*. The other events of interest are defined likewise. The media-player software is changed such that these events are published when a child process is initiated or killed.

```
1 eventtype ChildProcessEvent extends BaseEventType{
2  dynamiccontext:
3   PID : long;
4   parent : long;
5 }
6 event MPCoreInitiated instanceof ChildProcessEvent{}
7 event MPCoreKilled instanceof ChildProcessEvent{}
```

Listing 13. The specification of the event type ChildProcessEvent

In this example, the parent process is the only publisher of events of interest; consequently, the runtime environment of EventReactor is executed in the parent process by its main thread of execution. For each child process of interest, we extend the media-player software with a class that defines two methods: one for initiating the child process, and one for killing it. From within these methods, the events that are defined for the child process are published to the runtime environment of EventReactor. The media-player software is also changed such that the parent process invokes these methods when needed. Listing 14 shows an example of such a class for the child process *MPCore*, in which two methods `initMPCore` and `killMPCore` are defined.

```
1  public class MPCoreClass{
2      void initMPCore() {
3         ...
4         int processID = \\ create the child process MPCore
5         ...
6         MPCoreInitiated event = new MPCoreInitiated();
7         event.initializeDynamicAttribute("publisher", parentPID);
8         event.initializeDynamicAttribute("thread",CurrentThread.ID);
9         event.initializeDynamicAttribute("stacktrace",CurrentThread.stacktrace);
10        event.initializeDynamicAttribute("PID",processID);
11        event.initializeDynamicAttribute("parent", parentPID);
12        EventReactor.publish(event);
13     }
14     void killMPCore(){...}
15 }
```

Listing 14. An excerpt of *MPCoreClass*

Defining Auxiliary Information: As Figure 7 shows, the concern *AppProcess* has two attributes *init* and *kill*, which represent the methods that are used by the media-player software to create or kill a child process, respectively. The necessary information about these methods must also be defined in EventReactor. The compiler of EventReactor is extendable with new kinds of specifications, providing that suitable generators are provided to translate them into Prolog facts and queries. Using the feature of EventReactor, we provide a specification language to define the methods that must be invoked to construct or destroy a process. Listing 15 shows an excerpt of such specifications.

```
1 Method void initMPCore() In Class MPCoreClass
2 Method void killMPCore() In Class MPCoreClass
```

Listing 15. An excerpt of the specification to define methods

Defining Reactor Types: We provide `React` and `RestartProcess`, whose specification is depicted in Listing 16. The sole function of `React` is to publish a reactor event when it receives an event to process. The name of the reactor event must be provided as an argument to the reactor type. The reactor type `RestartProcess` restarts a group of child processes that is specified as the parameter of the reactor type. This reactor type publishes the reactor event `succeeded` of the type `RecoveryResult` if it successfully restarts a child process; otherwise, it publishes the reactor event `failed` of the type `RecoveryResult`.

```
1  reactortype React {
2      reaction = ReactClass;
3      events = { parameters.name : ReactorEventType};
4      parameters = {name : String};
5  }
6  reactortype RestartProcess {
7      reaction = RestartProcessClass;
8      events = { succeeded : RecoveryResult, failed: RecoveryResult};
9      parameters = {processes : List};
10 }
```

Listing 16. The specification of the reactor types

Defining the Concern *AppProcess*: Listing 17 defines the event package `MPCoreProcessPackage` in which necessary data are selected and event modules are defined to represent the child process *MPCore*. In lines 3–8, we select the events indicating construction and destruction of *MPCore*, and the methods whose execution causes the construction and destruction of this child process.

Lines 10–12 define the event module `MPCoreProcess` to represent the child process *MPCore*. Here, the events selected by `e_inited` and `e_killed` are grouped as the input interface of the event module. The reactor chain `AppProcessImpl` takes the selected events and methods as its arguments. As the output interface, the event module publishes the events `inited` and `killed`. Since there is only one child process as *MPCore* in the media-player software, the event module is specified to be instantiated in a singleton manner. The other child processes must be defined likewise.

Listing 18 defines the reactor chain `AppProcessImpl` implementing the functionality of the concern *AppProcess*. Here, the parameters `pinit` and `pkill` represent the methods that create or kill a child process, respectively. The parameters `pinited` and `pkilled` represent the events indicating that a child process is initiated or killed, respectively. The variables in the reactor chain correspond to the attributes of *AppProcess* in Figure 7.

```
1  eventpackage MPCoreProcessPackage{
2    selectors
3      e_inited = {E | isEventWithName(E, 'MPCoreInitiated')};
4      e_killed = {E | isEventWithName(E, 'MPCoreKilled')};
5      m_init = {M | isMethodWithName (M, 'initMPCore'),
6                    isClassWithName (C, 'MPCoreClass'), isDefinedIn(M, C)};
7      m_kill = {M | isMethodWithName (M, 'killMPCore'),
8                    isClassWithName (C, 'MPCoreClass'), isDefinedIn(M, C)};
9      eventmodules
10       MPCoreProcess := {e_inited, e_killed} <−
11                AppProcessImpl (m_init, m_kill, e_inited, e_killed) −>
12                {inited: ReactorEventType, killed: ReactorEventType};
13 }
```

Listing 17. Modular representation of *MPCore*

```
1  reactorchain AppProcessImpl (pinit:Method,pkill:Method,pinited:Event,pkilled:Event)
2  {
3    variables
4      init : Method = pinit;
5      kill : Method = pkill;
6      pid : Integer;
7      status : String;
8    reactors
9      reportInitiated : React = (event.name == pinited.name) =
10       { status = 'running'; pid = event.PID; reactor.name = 'inited'; };
11     reportKilled : React = (event.name == pkilled.name) =
12       { status = 'terminated'; pid = −1; reactor.name = 'killed';};
13 }
```

Listing 18. Implementing *AppProcesss*

Lines 9–10 define the reactor `reportInitiated` of type `React`, which only processes the input events represented by `pinited`. In the body of this reactor, reactor chain's attribute `status` is set to the value `''running''`, `pid` is assigned the unique identifier of the created process, and the value `''inited''` is specified as the name of the event that will be published by the reactor. The reactor `reportKilled` is defined analogously with the difference that no valid process ID exists.

At runtime, when the child process *MPCore* is created in the media player software, the event `MPCoreInitiated` is published. EventReactor will retrieve the singleton instance of the `AppProcessImpl` reactor chain and pass the event to the reactor `ReportInitiated`. Since the event is of interest, the reactor assigns the specified values to the variables defined in the reactor chain, and publishes the reactor event `inited`. Afterwards, the event `MPCoreInitiated` is received by the reactor `ReportKilled`, but the reactor ignores it because it is not of interest.

Defining the Concern *RecoveryUnit*: Listing 19 defines the event package `GRUnit` in which an event module is defined to represent a recovery unit for the global recovery of the media-player software. This recovery unit must group all the child processes, and it must report a failure when the child process *MPCore* is killed.

To this aim, lines 3–5 of Listing 19 select the event `killed` that is in the output interface of the event module `MPCoreProcess`, and name it as `e_killed` in the event package. Lines 6–9 select the event modules `MPCoreProcess`, `Audio`, `Video`, `UserInterface`. Lines 11–13 define the event module `GlobalRU` with the implementation `RecoveryUnitImpl`. The list of selected child processes (represented as event modules) is passed to the reactor chain `RecoveryUnitImpl`. The event module publishes a reactor event named `failure` as its output interface. Listing 20 defines a recovery unit for the local recovery of the child process *UserInterface*.

```
1  eventpackage GRUnit {
2    selectors
3      e_killed = {E | isEventWithName(E, 'killed'),
4                      isEventModuleWithName (EM, '*.MPCoreProcess'),
5                      isPublishedBy(E, EM)};
6      em_mpcore = {EM | isEventModuleWithName (EM, '*.MPCoreProcess')};
7      em_audio = ...
8      em_video = ...
9      em_ui = ...
10   eventmodules
11     GlobalRU := {e_killed} <-
12                   RecoveryUnitImpl ({em_mpcore, em_ui, em_audio, em_video})
13                   -> {failure : ReactorEventType};
14 }
```

Listing 19. Modular representation of the global recovery unit

```
1  eventpackage LRUnit{
2    selectors
3      e_uikilled = {E | isEventWithName(E, 'killed'),
4                        isEventModuleWithName (EM, '*.UIProcess'),
5                        isPublishedBy(E, EM)};
6      em_ui = {EM | isEventModuleWithName (EM, '*.UIProcess')};
7    eventmodules
8      LocalRU := {e_uikilled} <- RecoveryUnitImpl({em_ui})
9                     -> {failure : ReactorEventType};
10  }
```

Listing 20. Modular representation of a local recovery unit

Listing 21 defines the reactor chain `RecoveryUnitImpl` to implement the functionality of the concern *RecoveryUnit*. The reactor chain receives the list of corresponding child processes in its parameter `processLst` and stores them in the

variable `processes`. The reactor `reportFailure` is defined of type `React`. The name of the reactor event that will be published by `reportFailure` is specified as `'failure'`.

```
1 reactorchain RecoveryUnitImpl (processLst: List) {
2   variables
3     processes : List = processLst;
4   reactors
5     reportFailure : React = { reactor.name = 'failure'; };
6 }
```

Listing 21. Implementing *RecoveryUnit*

Defining the Concern *ProcessManager*: Listing 22 defines the event package `ProcessManagers`, in which event modules are defined to represent the concern *ProcessManager* of Recoverable Process. Lines 3–10 select the recovery units of interest and the events published by them. Lines 12–14 define the event module `RestartAll`. It takes the events selected by `e_global_failure` as input interface and binds the reactor chain `ProcessManagerImpl` to it. The value of the attribute `processes` that is defined in the implementation of `GlobalRU` is passed as the argument. As output, this event module publishes either `succeeded` or `failed`. The event module `RestartUI` is defined likewise in lines 15–17.

```
1  eventpackage ProcessManagers{
2    selectors
3      em_global = {EM | isEventModuleWithName (EM, '*.GlobalRU')};
4      em_local = {EM | isEventModuleWithName (EM, '*.LocalRU')};
5      e_global_failure = {E | isEventWithName(E, 'failure'),
6                             isEventModuleWithName (EM, '*.GlobalRU'),
7                             isPublishedBy(E, EM)};
8      e_local_failure = {E | isEventWithName(E, 'failure'),
9                            isEventModuleWithName (EM, '*.LocalRU'),
10                           isPublishedBy(E, EM)};
11   eventmodules
12     RestartAll := {e_global_failure} <-
13                             ProcessManagerImpl(em_global.processes)
14                             -> {succeeded: RecoveryResult, failed: RecoveryResult};
15     RestartUI := {e_local_failure} <-
16                            ProcessManagerImpl(em_local.processes)
17                             -> {succeeded: RecoveryResult, failed: RecoveryResult};
18 }
```

Listing 22. Modular representation of process managers

Listing 23 implements the functionality to restart a set of child processes that is specified by the parameter `processes`. At runtime, if the event `failure` is published by the event module `GlobalRU` and/or `LocalRU`, the reactor `Restart`

is informed of the event, and restarts the child processes that form the recovery unit. For this matter, the reactor retrieves the necessary information about the methods that kill and re-initialize a child process from the attributes `init` and `kill` of the corresponding instance of `AppProcessImpl`, and invokes them.

```
1 reactorchain ProcessManagerImpl(processes: List){
2   reactors
3     restart : RestartProcess = {reactor.processes=processes};
4 }
```

Listing 23. Implementing *ProcessManager*

Defining an Application-Specific Composition Strategy: The event modules in Listings 19 and 20 both specify the child process *UserInterface* as an element of their recovery unit. Assume that at runtime, the child process *MPCore* fails. As a consequence the global recovery kills and re-initializes the child processes *Audio*, *Video*, *UserInterface*. When the child process *UserInterface* is killed for the global recovery, the event `failure`, which is specified in lines 8–10 of Listing 22, is detected and the child process *UserInterface* is re-initialized for the local recovery. As a result, there will be two processes running as *UserInterface*.

To overcome the above problem, we want to specify a composition of global and local recovery with each other with the following semantics: If global recovery is being executed on a group of processes, these processes must not be recovered locally. Listing 24 defines the event package `RecoveryConstraint` in which this constraint is specified. The desired constraints among the event modules `RestartAll` and `RestartUI` are defined in the part `constraints` of the event package. In line 6, the operator `ignore`, which is a predefined composition operator in EventReactor, indicates that the event module `em_RestartUI` must ignore the events that are published during the execution of the event module `em_RestartAll`.

```
1 eventpackage RecoveryConstraint{
2   selectors
3     em_restartAll = {EM | isEventModuleWithName (EM, '*.RestartAll')};
4     em_restartUI = {EM | isEventModuleWithName (EM, '*.RestartUI')};
5   constraints
6     ignore(em_restartUI, em_restartAll);
7 }
```

Listing 24. Representing recovery constraints

More complex semantics can be defined as event modules, and dedicated reactor types can be defined for their implementation. For example, assume that if the global recovery fails to re-initialize *UserInterface*, the local recovery must still try to do so. Listing 25 defines the event module `Coordinator` for

this matter. As explained before, the reactor type `RestartProcess` publishes the event `succeeded` if it successfully restarts a process; otherwise it publishes the event `failed`. The name of the child process is maintained in the attribute `processName` of these events. Lines 3–5 select the event `failed` that is published by the event module `RestartAll`. Line 6 selects the event module `UIProcess`. Line 8 defines the event module `Coordinator` with `e_failure` as its input interface, and the reactor chain `CoordinatorImpl` as its implementation. The argument `‘‘UserInterface’’` of the reactor chain indicates the name of the child process of interest, and the argument `em_ui` represents the child process.

Listing 26 defines the reactor chain `CoordinatorImpl`, which receives the name of the child process of interest and a reference to the event module representing it as its parameters. In the body of the reactor chain, we reuse the reactor type `RestartProcess` to define the reactor `restart`. This reactor only processes those events whose attribute `processName` is equal to `failedProcess`. In the body of the reactor, the parameter `process` is assigned to the parameter `processes` of the reactor. At runtime, if the event module `RestartAll` publishes the event `failed` for the child process *UserInterface*, the event module `coordinator` will be instantiated, and the reactor chain `CoordinatorImpl` will restart the child process.

```
1 eventpackage Coordination{
2   selectors
3     e_failure = {E | isEventWithName (E, 'failed'),
4                      isEventModuleWithName (EM, '*.RestartAll'),
5                      isPublishedBy(E, EM)};
6     em_ui = {EM | isEventModuleWithName (EM, '*.UIProcess')};
7   eventmodules
8     Coordinator := {e_failure}<- CoordinatorImpl ('UserInterface', em_ui)->{ };
9 }
```

Listing 25. Modular representation of the application-specific composition strategy

```
1 reactorchain CoordinatorImpl (failedProcess, process) {
2   reactors
3     restart : RestartProcess = (event.processName ==failedProcess)
4       {reactor.processes = {process};}
5 }
```

Listing 26. Implementing the application-specific composition strategy

7 Performance Evaluation

Inevitably, EventReactor imposes overhead on the base program that is interacting with it. The sources of this overhead are mainly: a) the operation to match a published event with an event defined in EventReactor, b) the operation to retrieve the selectors that match the event, c) the operation to lookup the corresponding instance of event modules or instantiate the event module if there is

no instance available, and d) the operation to process the event by the corresponding reactors. As explained earlier in this section, in the current version of EventReactor, selectors can only query events based on their static attributes. Since, the values of these attributes are known at compile time, the matching between selectors and events is performed at compile time, and the compiler provides the information about this match for the runtime environment. This helps to reduce the runtime overhead imposed by EventReactor. The time to process the event by reactors depends on the complexity of the function performed by the reactors.

To evaluate the runtime overhead added by EventReactor, we ran two example scenarios on a 2.00 GHz Intel Core 2, with 3GB RAM running the JVM version 1.6.0 under Mac OS X 10.6.8. The first scenario illustrates how the runtime overhead varies if the number of events to be processed varies. We assume that there is one thread of execution in which the events *login*, *save*, *open*, *write*, *close*, and *logout* are published and their order of occurrence is verified using the event module *Verification*. We assume that the number of times that this sequence of events occurs varies from 1 to 500 times, which means publishing 6 to 30,000 events. We executed this scenario 6 times; to avoid the initialization overhead that is imposed in the first iteration, we discarded the results of the first execution. Figure 9 shows the average measured time to process the events, which has a linear growth with the goodness of fit $R^2 = 0.99964$. Here, the X-axis and Y-axis show the number of events and the processing time in milliseconds, respectively.

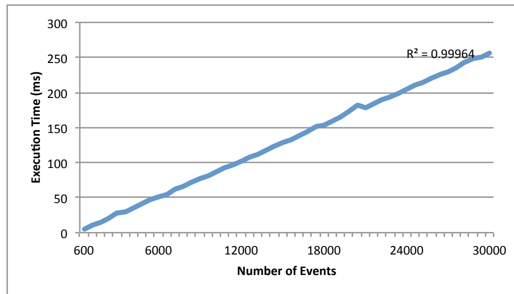


Fig. 9. The runtime overhead in processing events in a single-threaded case

In our running example, distinct instances of the event module *Verification* must be created for each thread of execution in which the specified events are published. To measure the overhead imposed to lookup instances of this event module, in the second scenario, we assume that there are multiple threads of executions in which the aforementioned sequence of events occurs 500 times. Again, we executed this scenario 6 times and show the average of the last 5 runs

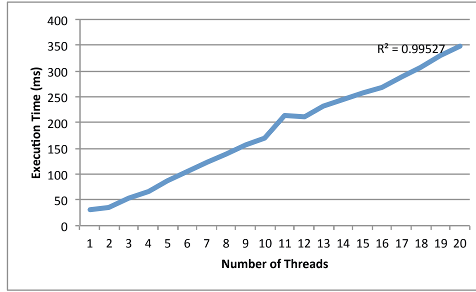


Fig. 10. The runtime overhead in processing events in a multi-threaded case

in Figure 10. Here, the X-axis and Y-axis show the number of threads and the processing time in milliseconds, respectively. This measurement shows that the time for processing the events is also linear in the number of threads, with the goodness of fit $R^2 = 0.99527$.

8 Discussion

As for the theoretical aspects of runtime verification, it is important to provide suitable means for industrial programmers to implement and adopt RV techniques. To this aim, various RV DSLs have been introduced in the literature. To implement RV techniques at an abstraction level that is close to the domain of interest, and to increase the comprehensibility and the reusability of implementations, we claimed that the abstractness, modularity, and compose-ability requirements must be fulfilled in the implementations. According to the comparison provided in this paper, the current RV DSLs fall short in this matter.

To prevent designing and implementing an RV DSL from scratch, this paper discussed that there is an inevitable need for a language composition framework, which provides the necessary means to define new DSLs, and to implement domain-specific crosscutting concerns (e.g. the RV concerns) in a modular and compose-able way. This paper identified four requirements that must be fulfilled by such a framework, explained Event Composition Model as a base model for such a framework, and discussed its implementation in the EventReactor language.

In Event Composition Model, event modules and events are means for the modularization and composition of concerns, respectively. A set of predefined event types and events is provided, where user-defined ones are also supported. This has been shown in Listing 4 and 13. Necessary information can be abstracted from the publisher via event attributes, which are also programmable. Individual event types and events are defined modularly; this facilitates defining reusable libraries of them.

Primitive information can be observed and abstracted from the base software through Prolog queries. To increase the reusability of specifications, these queries can be defined in separate event packages. More complex observation semantics

can be defined over such primitive information via event modules. Since events can be observed from the base software implemented in various languages, it is possible to define the specifications of RV concerns abstractly from the implementation of the base software in a modular way. This is shown in Listings 6 and 7.

We represented that individual verification concerns and actions can be programmed and modularized via event modules. By means of events, these modules can flexibly be composed and their composition constraints can be programmed and modularized as event modules; this is shown for example in Listing 25. In both examples in this paper, we showed that due to the uniform representation of RV concerns as event modules, there is no limit in the number of levels in the event module hierarchy. This provides the necessary basis to implement hierarchal adaptive techniques.

Where it is possible to define libraries of reusable event modules, the reusability is increased further by separating the specification of reactor chains from event modules. This facilitates defining libraries of reactor chains and reusing them as the implementation of various event modules; Listing 17 shows an example.

A finer-grained modularity is achieved by means of domain-specific reactor types, which modularize the implementation of individual verification concerns and/or actions. Listings 23 and 26 show an example reuse of the reactor type `RestartProcess`. The reusability is increased further by supporting parametric reactor chains and reactor types.

The kinds of actions that can/must be applied at runtime is being studied in both runtime verification [29] and self-adaptive communities [30]. The kinds of actions cannot be anticipated, and are in general domain-specific. For example, in [29], a study is performed on the security domain, which results in four kinds of automaton named as: *truncation automata* which can only terminate applications, *suppression automata* which can terminate applications and suppress individual actions, *insertion automata* which can terminate and insert, and *edit automata* which combines the powers of suppression and insertion automata. In the domain of energy optimization, others kinds of actions such as changing the operational states of hardware devices can be considered.

Each of the above-mentioned actions require a dedicated focus, to identify their contextual interaction and algorithms; this is out of scope of this paper. Nevertheless, by means of our illustrative examples, we discussed how the actions that are typically considered by the current RV DSLs can be programmed in EventReactor via reactor types. For example, the reactor type `ForceReturn` implements the functionality to suppress the invocation or execution of an individual method. The reactor type `RestartProcess` can be regarded as an implementation of the edit automata, in which the execution of a child process is suppressed by destroying the process, and is afterwards restarted by invoking a method. The necessary contextual information can be specified and queries via event attributes, auxiliary information, and/or via event modules. For example, as Listing 22 shows, the necessary information about the child

processed that must be restarted is provided as arguments to the reactor chain `ProcessManagerImpl`; these arguments refer to the instances of the event module `AppProcess` representing the child processes. The necessary information about the methods that must be invoked for the construction and destruction of the child processes, is defined as the auxiliary information in Listing 15, and is maintained in the variables `init` and `kill` (see Listing 18) of the corresponding instances of `AppProcess`.

9 Future Work

The EventReactor language provides dedicated constructs to implement the concepts of Event Composition Model. In the version presented in this paper, events can be queried based on their static attributes. Nevertheless, in the same way, it is possible to extend the query language to facilitate event selection based on the dynamic attributes. Supporting this feature while maintaining low runtime overhead is our future direction.

Runtime verification of parametric properties is in general a challenging issue [13, 4]. There are various strategies to deal with parameter binding in the specifications [13]; one example is to associate individual instances of verification modules to each individual group of correlated events. The correlation information can be abstracted from the base software via the attributes of the events. By means of programmable event attributes and programmable instantiation strategy for events modules, EventReactor provides the necessary basis to support parametric properties. For example, one may define a dedicated attribute in the events of interest, which has the same value in each set of correlated events. Accordingly, as for the instantiation strategy of the event module, one can specify that distinct instances of the event module must be created for each distinct value of this attribute.

In the literature, various kinds of correlations are studied, whose detection can be automated by the compiler. Examples are the correlation between a constructor and constructed objects, the correlation between a composite object and its part objects. As future work, we would like to extend both the event specification language and the compiler of EventReactor to express these correlations and automate their detection. In addition, we would like to evaluate the suitability of EventReactor for verifying a set of parametric properties, which are studied in the literature.

Since there is an analogy between the concepts of Event Composition Model and the ones in the aspect-oriented languages, the EventReactor language can be regarded as a language composition framework for aspect-oriented DSLs. There are already several aspect-oriented language composition frameworks [31, 32], with the similar goals as Event Composition Model. However, they fall short in supporting some or all of our requirements open-ended kinds of events, open-ended kinds of DSLs and base languages, and open-ended kinds of composition strategies. Consequently, the abstractness, modularity, and compose-ability requirement cannot be achieved in the implementation of RV concerns through

these frameworks. As future work, we will investigate more along this line, and employ Event Composition Model and EventReactor as the base for implementing aspect-oriented DSLs.

The set of predefined events, which can automatically be detected in the base software, is currently limited to method-based events. Some aspect-oriented languages already facilitate detecting a larger set of events, such as occurrence of exception. The same techniques can be adopted in EventReactor language, which we consider as future work.

Event Composition Model does not fix possible implementation of its concepts. For example, in the current implement of EventReactor, reactors are composed with each other within a reactor chain in a sequential manner. A language may also implement parallel composition of reactors. Likewise, a language may support more complex predicate-based instantiation strategy of event modules. As future work, we would like to extend EventReactor to support other alternative implementation of the concepts of Event Composition Model.

We would also like to adopt the EventReactor language to implement other kinds of techniques, such as self-adaptive software systems, which have similar characteristics as RV techniques. Implementing energy-optimization concerns [33] and necessary adaptation actions are our current focus.

Acknowledgements. The authors thank Dr. Christoph Bockisch and the anonymous reviewers for their valuable feedbacks on this paper.

References

1. Khurshid, S., Sen, K. (eds.): RV 2011. LNCS, vol. 7186. Springer, Heidelberg (2012)
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
3. Easwaran, A., Kannan, S., Sokolsky, O.: Steering of Discrete Event Systems: Control Theory Approach. Electron. Notes Theor. Comput. Sci. 144, 21–39 (2006)
4. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An Overview of the MOP Runtime Verification Framework. International Journal on Software Techniques for Technology Transfer, 249–289 (2011)
5. Havelund, K.: Runtime verification of C programs. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 7–22. Springer, Heidelberg (2008)
6. Bartetzko, D., Fischer, C., Moller, M., Wehrheim, H.: Jass - Java with Assertions. Electronic Notes in Theoretical Computer Science 55(2), 1–15 (2001)
7. Drusinsky, D.: The Temporal Rover and the ATG Rover. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 323–330. Springer, Heidelberg (2000)
8. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview, pp. 49–69. Springer (2004)
9. Rosenblum, D.S.: Towards a Method of Programming with Assertions. In: Proceedings of the 14th International Conference on Software Engineering, ICSE 1992, Melbourne, Australia, pp. 92–104. ACM (1992)

10. Malakuti, S., Bockisch, C., Akşit, M.: Applying the Composition Filter Model for Runtime Verification of Multiple-Language Software. In: Proceedings of the 20th IEEE International Conference on Software Reliability Engineering, ISSRE 2009, pp. 31–40. IEEE Press, Piscataway (2009)
11. Martin, M., Livshits, B., Lam, M.S.: Finding Application Errors and Security Flaws Using PQL: A Program Query Language. SIGPLAN Not. 40, 365–383 (2005)
12. Bauer, L., Ligatti, J., Walker, D.: Composing Expressive Runtime Security Policies. ACM Trans. Softw. Eng. Methodol. 18 (2009)
13. Pavel, C.A., Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Moor, O.D., Sereni, D., Sittampalam, G., Tibble, J.: Adding Trace Matching with Free Variables to AspectJ. In: OOPSLA, pp. 345–364 (2005)
14. Barringer, H., Havelund, K.: TraceContract: A scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
15. Malakuti, S.: Event Composition Model: Achieving Naturalness in Runtime Enforcement. PhD thesis, University of Twente (2011)
16. Malakuti, S., Akşit, M.: Evolution of Composition Filters to Event Composition. In: Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC 2012, pp. 1850–1857. ACM (2012)
17. Parnas, D.L.: On the Criteria To Be Used in Decomposing Systems into Modules. Communications of the ACM 15, 1053–1058 (1972)
18. Akşit, M.: Separation and Composition of Concerns. ACM Computing Surveys 28 (1996)
19. Sozer, H., Abreu, R., Akşit, M., van Gemund, A.J.: Increasing System Availability with Local Recovery Based on Fault Localization. In: International Conference on Quality Software, pp. 276–281 (2010)
20. Sozer, H.: Architecting Fault-Tolerant Software Systems. PhD thesis, University of Twente (2009)
21. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
22. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
23. AspectC, `http://www.cs.ubc.ca/labs/spl/projects/aspectc.html`
24. Steimann, F., Pawlitzki, T., Apel, S., Kästner, C.: Types and Modularity for Implicit Invocation with Implicit Announcement. ACM Trans. Softw. Eng. Methodol. 20(1), 1:1–1:43 (2010)
25. de Roo, A., Hendriks, M., Havinga, W., Durr, P., Bergmans, L.: Compose*: A Language- and Platform-Independent Aspect Compiler for Composition Filters. In: International Workshop on Academic Software Development Tools and Techniques (2008)
26. Hopcroft, J.E., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley (2000)
27. Malakuti, S., Akşit, M.: Event-Based Modularization of Reactive Systems. In: Concurrent Objects and Beyond. LNCS (2013) (to appear)
28. Riordan, M.: Designing Relational Database Systems. Microsoft Press (1999)
29. Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. International Journal of Information Security 4, 2–16 (2005), doi:10.1007/s10207-004-0046-8
30. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)

31. Havinga, W., Bergmans, L., Akşit, M.: Prototyping and Composing Aspect Languages: Using an Aspect Interpreter Framework. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 180–206. Springer, Heidelberg (2008)
32. Tanter, É.: An Extensible Kernel Language for AOP. In: Proceedings of the Workshop on Open and Dynamic Aspect Languages (ODAL) (2006)
33. Malakuti, S., te Brinke, S., Bergmans, L., Bockisch, C.: Towards Modular Resource-Aware Applications. In: Proceedings of the 3rd International Workshop on Variability & Composition (VariComp 2012), pp. 13–17. ACM, New York (2012)

Method Slots: Supporting Methods, Events, and Advices by a Single Language Construct

YungYu Zhuang and Shigeru Chiba

The University of Tokyo
http://www.csg.ci.i.u-tokyo.ac.jp/

Abstract. To simplify the constructs that programmers have to learn for using paradigms, we extend methods to a new language construct, a *method slot*, to support both the event-handler paradigm and the aspect paradigm. A *method slot* is an object's property that can keep more than one function closure and be called like a method. We also propose a Java-based language, *DominoJ*, which replaces methods in Java with *method slots*, and explains the behavior of *method slots* and the operators. Then we evaluate the coverage of expressive ability of *method slots* by comparing *DominoJ* with other languages in detail. The feasibility of *method slots* is shown as well by implementing a prototype compiler and running a preliminary microbenchmark for it.

Keywords: aspect-oriented programming, event-driven programming.

1 Introduction

The event-handler paradigm has been recognized as a useful mechanism in a number of domains such as user interface, embedded systems, databases [33], and distributed programming. The basic idea of the event-handler paradigm is to register an action that is automatically executed when something happens. At first it was introduced as techniques and libraries [7,29,27] rather than supported at language level. Recently, supporting it at language level is a trend since a technique such as the Observer pattern [7] cannot satisfy programmers' need. The code for event triggers and observer management scatters everywhere. To address the issues, supporting events by a language construct is proposed in a number of languages [17,3,22,6,13,9]. Implicit invocation languages [8] might be classified into this category.

On the other hand, the aspect paradigm [14] is proposed to resolve crosscutting concerns, which cannot be modularized by existing paradigms such as object orientation. Although the aspect paradigm and the event-handler paradigm are designed for different scenarios, the constructs introduced for them are similar and can work as each other from a certain point of view.

In order to simplify the language constructs programmers have to learn, we borrow the idea of slots from Self [31] to extend the method paradigm in Java. In Self, an object consists only of slots [24], which may contain either a value or a method. In other words, there is no difference between fields and methods since a

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 70–108, 2014.
© Springer-Verlag Berlin Heidelberg 2014

method is also an object and thus can be kept in a field. We extend the slot and bring it to Java-like languages by proposing a new language construct named *method slot*. A *method slot* is an object's property that can keep more than one closure at the same time. We also present a Java-based language named *DominoJ*, where all methods in plain Java are replaced with *method slots*, to support both the event-handler paradigm and the aspect paradigm.

Our contributions presented in this paper[1] are two fold. First, we propose a new language construct, a *method slot*, to extend the method paradigm. Second, we introduce *method slots* to a Java-based language named *DominoJ*, and demonstrate how to use for the event-handler paradigm and the aspect paradigm.

2 Motivation

With the evolution of software, more and more programming paradigms are developed for various situations. During programmers' life, they are always learning new paradigms and thinking about which ones are most suitable for the job at hand. For example, the event-handler paradigm is widely adopted by GUI frameworks [32,18,25]. When we write GUI programs with modern GUI libraries, we usually have to write a number of handlers for different types of events. The AWT [25] of Java is a typical example. If we want to do something for mouse events occurring on a button, we have to prepare a mouse listener that contains handler methods for those mouse events, and register the listener to the specified button object. A GUI program can be regarded as a composite of visual components, events, and handlers. The visual components and handlers are main logic, and events are used for connecting them. Indeed we have been familiar with using the event-handler paradigm for GUI programs, but it is far from our first "hello world" program. We are told to carefully consider the total execution order when users' input is read. If the event-handler paradigm is used, we can focus on the reaction to users' input rather than the order of users' input. Whether the mouse is clicked first or not does not matter. Another example is the aspect paradigm. Aspect-oriented programming is developed to modularize crosscutting concerns such as logging, which cannot be modularized by using only object-oriented programming. With the aspect paradigm, crosscutting concerns can be gathered up in an aspect by advices. At the same time, programmers cannot check only one place for understanding the behavior of a method call since advices in other places are possibly woven together. It also takes effort to get familiar with the aspect paradigm since it is quite different from our other programming experience.

To use a paradigm, just learning its concept is not enough. After programmers got the idea of a paradigm, they still have to learn new language constructs for the paradigm. Some paradigms like the aspect paradigm are supported with dedicated language constructs since the beginning because they cannot be represented well by existing syntax. On the other hand, although other paradigms like the event-handler paradigm have been introduced at library level for a long

[1] This paper is an extension to the one we presented at Modularity:AOSD2013.

time, there are still good reasons for reintroducing them with direct support at language level [17,22,9]. Maybe one reason is that events are complicated in particular when we are not users but designers of a library. Besides GUI libraries, the event-handler paradigm is also implemented in a number of libraries for several domains such as simple API for XML [30] and asynchronous socket programming. Some techniques such as the Observer pattern [7] used in those libraries cannot satisfy the needs of defining events and tend to cause code scattering and tangling. Supporting paradigms by language constructs is a trend since it makes code more clear and reusable. Furthermore, a language supported paradigm may have associated static checks.

However, learning language constructs for a paradigm is never easy, especially for powerful paradigms like the aspect paradigm. Moreover, the syntax is usually hard to share with other paradigms. Even though programmers got familiar with the language constructs for a paradigm, they still have to learn new ones for another paradigm from the beginning. Given that all language constructs we need can be put into a language together, they look too complex and redundant. How to pick up the best language to implement a program with all the required paradigms is always a difficult issue. This motivates us to find out an easy, simple, and generic language construct supporting multiple paradigms.

If we look into the language constructs for the event-handler paradigm and the aspect paradigm, there is a notable similarity between them. Both of them introduce a way to define the effect of calling specified methods. The differences are where the reactions are and what the reactions are targeted at. Listing 1 is a piece of code in EScala[2] [9], which is a typical event mechanism, showing how to define a moved event for the setPosition method in the Shape class. Here we specify that refresh method on a Display object should be executed after setPosition method is executed. As shown in Listing 2, the reaction can also be represented in AspectJ [26], the most well-known aspect-oriented language.

By comparing the two pieces of code, we can find that pointcuts are close to events and advices can work as the += operator for handlers. They both refresh the display when the specified method is executed, but there is a significant difference between them. In EScala version, one Display object is mapped to one Shape object and the refresh action is performed within the Shape object. On the other hand, in AspectJ version there is only one Display object in the whole program and the refresh action is in UpdateDisplay, which is completely separated from Display and Shape. From the viewpoint of the event-handler paradigm, such behavior is an interaction between objects, so the reaction is defined inside the class and targeted at object instances; the encapsulation is preserved. From the viewpoint of the aspect paradigm, it is important to extract the reaction for the obliviousness since it is a different concern cutting across several classes. So the reactions are grouped into a separate construct and targeted at the class. Although the two paradigms are developed from different points of view, the language constructs used for them are quite similar. Furthermore, both the paradigms depend on the most basic paradigm, the method paradigm,

[2] The syntax follows the example in EScala 0.3 distribution.

Listing 1. Defining a reaction in EScala

```
1  class Display() {
2    def refresh() {
3      System.out.println("display is refreshed.")
4    }
5  }
6  class Shape(d: Display) {
7    var left = 0; var top = 0
8    def setPosition(x: Int, y: Int) {
9      left = x; top = y
10   }
11   evt moved[Unit] = afterExec(setPosition)
12   moved += d.refresh
13 }
14 object Test {
15   def main(args: Array[String]) {
16     val d = new Display()
17     val s = new Shape(d)
18     s.setPosition(0, 0)
19   }
20 }
```

since both events and pointcuts cause the execution of a method-like construct. This observation led us to extend the method paradigm to support both the event-handler paradigm and the aspect paradigm. To a programmer, there are too many similar language constructs for different paradigms to learn, so we assume that the integration and simplification are always worth doing.

3 DominoJ

We extend methods to a new language construct named a *method slot*, to support methods, events, and advices. We also show our prototype language named *DominoJ*, which is a Java-based language supporting *method slots* and fully compatible with plain Java.

3.1 Method Slots

Although methods and fields are different constructs in several languages such as C++ and Java, there is no difference between them in other languages like JavaScript. In JavaScript, a method on an object (strictly speaking, a function closure) is kept and used as other fields. Figure 1 shows a Shape object s, which has two fields: an integer field named x and a function field named setX. We use the following notation to represent a closure:
$\langle$*return type*$\rangle$ ($\langle$*parameter list*$\rangle$) -> { $\langle$*statements*$\rangle$ }
  | $\langle$*variable binding list*$\rangle$
where the variable binding list binds nonlocal variables in the closure. The value stored in field setX is a function closure whose return type and parameter type are void and (int), respectively. The variable this used in the closure is bound to s given by the execution context. When we query the field by s.setX, the function

Listing 2. Defining a reaction in AspectJ

```
1  public class Display {
2    public static void refresh() {
3      System.out.println("display is refreshed.");
4    }
5  }
6  public class Shape {
7    private int left = 0; private int top = 0;
8    public void setPosition(int x, int y) {
9      left = x; top = y;
10   }
11 }
12 public aspect UpdateDisplay {
13   after() returning:
14    execution(void Shape.setPosition(int, int)) {
15       Display.refresh();
16   }
17 }
18 public class Test {
19   public static void main(String[] args) {
20     Shape s = new Shape();
21     s.setPosition(0, 0);
22   }
23 }
```

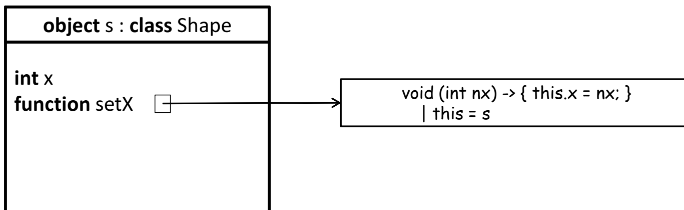


Fig. 1. In JavaScript, both an integer and a function are fields on an object

closure is returned. When we call the field by s.setX(10), the function closure is executed.

We extend this field in JavaScript to keep an array of function closures rather than just one function closure. As shown in Figure 2, the extended field named a *method slot* can keep more than one function closure. DominoJ replaces a method with a method slot in plain Java. All method-like declarations and calls are referred to method slots. A method slot is a closure array and is an object's property like a field. Like functions or other fields, method slots are typed and statically specified when they are declared. The type of method slot includes its return type and parameter types. All closures in it must be declared with the same type.

Listing 3 shows a piece of sample code in DominoJ. It looks like plain Java, but here setX is a method slot rather than a method. The syntax of method slot declaration is shown below:

$\langle modifier\rangle^*$ $\langle return\ type\rangle$ $\langle identifier\rangle$ "(" $\langle parameter\ list\rangle$? ")" $\langle throws\rangle$?
  ($\langle default\ closure\rangle$ | ";")

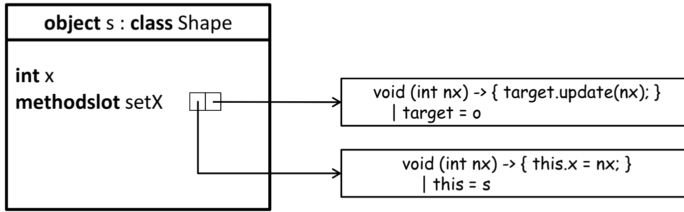


Fig. 2. A method slot is an extended field that can keep more than one function closure

The default closure is similar to the method body in Java except it is optional. The modifiers can be public, protected, or private for specifying the visibility of the method slot. This ensures that the access to the method slot can be controlled as the methods in plain Java. The modifier static can be specified as well. Such static method slots are kept on the class objects so can be referred using the class name like calling the static method in plain Java. The modifier abstract can also be used to specify that the method slot should be implemented by the subclasses. A method slot can be another kind of "abstract" by being declared without a default closure:

```
public void setX(int nx);
```

Unlike the modifier abstract, this declaration means that the method slot is an empty field and its behavior depends on the closures added to it later. In Listing 3, the method slot setX has a default closure, so the following function closure will be created and inserted into setX automatically when a Shape object, s, is instantiated:

```
void (int nx) -> { this.x = nx; }
  | this = s
```

Now there is only one closure in the method slot setX. If we add another closure to setX, the object may look like the s object in Figure 2. How to add such a closure to a method slot will be demonstrated in the next subsection.

A method slot can also be declared with the modifier final to specify that it cannot be overridden in the subclasses. Although fields are never overridden in either prototype-based languages like JavaScript or class-based languages like Java, method slots can be overridden in subclasses. Declaring a method slot with the same signature overrides but does not hide the one in the superclass. When a method slot is queried or called on an object, the overriding method slot is selected according to the actual type of the object. It is also possible to access the overridden method slot in the superclass through the keyword super. Note that method slots must be declared within a class and cannot be declared as local variables. Thus the usage of this and super in the default closure are the same as in a Java method, which refer to the owning class and its superclass, respectively. Constructors are method slots as well, and super() is allowed since it calls the overridden constructor.

Listing 3. A sample code in DominoJ

```
public class Shape {
  private int x;
  public void setX(int nx) {
    // default closure
    this.x = nx;
  }
}
public class Observer {
  private int count;
  public void update(int i) {
    this.count++;
  }
  public static void main(String[] args) {
    Shape s = new Shape();
    Observer o = new Observer();
    s.setX += o.update;
    s.setX(10);
  }
}
```

Listing 4. The algorithm of calling a method slot

```
; call a methodslot
(define (call-methodslot object slotname args)
  (let* ((methodslot (get-field object slotname (get-type args)))
         (return_type (get-return-type methodslot)))
    (let execute-closures ((closures (get-closures methodslot))
                           ($retval (cond ((boolean? return_type) #f)
                                          ((number? return_type) 0)
                                          (else '()))))
      (if (null? closures)
        $retval
        (let (($retval (execute-a-closure (car closures) args)))
          (execute-closures (cdr closures) $retval))))))
```

When a method slot is called by () operator, the closures in it are executed in order. The arguments given to the method slot are also passed to its closures. The return value returned by the last closure is passed to the caller (if it is not the void type). A closure can use a keyword $retval to get the return value returned by the preceding closure in the method slot. If the closure is the first one in the method slot, $retval is given by a default value (0, false, or null). If the method slot is empty, the caller will get the default value and no exception is thrown. It is reasonable since the empty state is not abnormal for an array and just means that nothing should be done for the call at that time. The behavior of a method slot can be dynamically modified at runtime, while still statically typed and checked at compile time. How to call a method slot is described in Scheme as shown in Listing 4.

Table 1. The four operators for method slots

Operator	Description
=	*add a new function closure and remove the others from the method slot.*
^=	*insert a new function closure at the beginning of the array.*
+=	*append a new function closure to the end of the array.*
-=	*remove function closures calling the method slot at the right-hand side.*

3.2 Operators for Method Slots

DominoJ provides four operators for manipulating the closures in a method slot: =, ^=, +=, and -=, as shown in Table 1. These operators are borrowed from C# and EScala, and are the only different syntax from Java. It is possible to add and remove a function closure to/from a method slot at runtime.

Their operands at both sides are method slots sharing the same type. Those operators except -= create a new function closure calling the method slot at the right-hand side, and add it to the method slot at the left-hand side. The method slot called by the function closure will get the same arguments which are given to the method slot owning the function closure. In other words, a reference to the method slot at the right-hand side is created and added to the method slot at the left-hand side. The syntax of using the operators to bind two method slots is shown below:

$\langle expr \rangle$"."$\langle methodslot \rangle$ $\langle operator \rangle$ $\langle expr \rangle$"."$\langle methodslot \rangle$";"

where $\langle expr \rangle$ can be any Java expression returning an object, or a class name if the following $\langle methodslot \rangle$ is static. When the binding statement is executed at runtime, the $\langle expr \rangle$ at both sides will be evaluated according to current execution context and then given to the operator. In other words, the $\langle expr \rangle$ at the right-hand side is also determined at the time of binding rather than the time of calling. The object returned by the $\langle expr \rangle$ at the left-hand side helps to find out the method slot at the left-hand side, where we want to add or remove the new function closure. The object got by evaluating the $\langle expr \rangle$ at the right-hand side is attached to the new function closure as a variable target, which is given to the new function closure along with the execution context at the time of calling. For example, the binding statement in Line 16 of Listing 3 creates a new function closure calling the method slot update on the object o by giving target = o, and appends it to the method slot setX on the object s.

```
void (int nx) -> { target.update(nx); }
  | target = o
```

Then the status of the s object will be the same as the one shown in Figure 2. When the slot setX on the object s is called as Line 17 in Listing 3, the default closure and the slot update on the object o are sequentially called with the same argument: 10. Note that all closures in a method slot get the same execution context except the side effects caused by the preceding closures in the array of that method slot, where this refers to the object owning the method slot, and therefore, the callee method slot in target must be accessible from the caller

Listing 5. The algorithms of the four operators

```
1  ; operator =
2  (define (assign-closure methodslot object slotname)
3    (let ((closure `(call-methodslot ,object ,slotname args)))
4      (set-closures methodslot closure)))
5
6  ; operator ^=
7  (define (insert-closure methodslot object slotname)
8    (let ((closure `(call-methodslot ,object ,slotname args)))
9      (set-closures methodslot (append closure (get-closures methodslot)))))
10
11 ; operator +=
12 (define (append-closure methodslot object slotname)
13   (let ((closure `(call-methodslot ,object ,slotname args)))
14     (set-closures methodslot (append (get-closures methodslot) closure))))
15
16 ; operator -=
17 (define (remove-closure methodslot object slotname)
18   (let ((closure `(call-methodslot ,object ,slotname args)))
19     (set-closures methodslot (remove (lambda (x) (equal? x closure))
20                                      (get-closures methodslot)))))
```

method slot in this. With proper modifiers, a method slot cannot call and be called without any limitation. The behavior avoids breaking the encapsulation in object-oriented programming.

The -= operator removes function closures calling the method slot at the right-hand side from the method slot at the left-hand side. It is also possible to remove the default closure from a slot by specifying the same method slots at both sides:

```
s.setX -= s.setX;
```

Operators manipulate the default closure only when the method slots at both sides are the same one, otherwise operators regard the right-hand side as a closure calling that method slot. Note that the default closure is never destroyed even when it is removed. The algorithms of the four operators are described in Scheme in Listing 5.

Although a method slot at the right operand of the operators such as += must have the same type that the left operand has, there is an exception. If a method slot takes only one parameter of the Object[] type and its return type is Object or void, then it can be used as the right operand whatever the type of the method slot at the left operand is. Such a method slot can be used as a generic method slot. The type conversion when arguments are passed is implicitly performed. Listing 6 shows how to check the type of two method slots in Scheme.

DominoJ allows binding method slots to constructors by specifying class name instead of the object reference and giving the keyword constructor as the method slot at the left-hand side. For example,

```
Shape.constructor += Observer.init;
```

means that creating a closure calling the static method slot init on the class object Observer and appending to the constructor of Shape. Here the return

Listing 6. The algorithm of checking the types

```
1  ; is same type
2  (define (same-type? l_methodslot r_methodslot)
3    (and (equal? (get-return-type l_methodslot)
4                 (get-return-type r_methodslot))
5         (equal? (get-parameter-types l_methodslot)
6                 (get-parameter-types r_methodslot))))
7
8  ; is generic type
9  (define (generic-type? l_methodslot r_methodslot)
10   (and (equal? (get-parameter-types r_methodslot)
11                "Object[]")
12        (if (equal? (get-return-type l_methodslot)
13                    "void")
14          (equal? (get-return-type r_methodslot)
15                  "void")
16          (equal? (get-return-type r_methodslot)
17                  "Object"))))
```

type of init should be void, and the parameter types must be the same as the constructor. Note that the closures appended to the constructor cannot block the object creation. This design ensures that the clients will not get an unexpected object, but additional objects can be created and bound to the new object. For example, in the default closure of init, an instance of Observer can be created and its update can be bound to the method slot setX of the new Shape object. Using constructor at the right-hand side is not allowed.

Since Java supports method overloading, some readers might think the syntax of method slots have ambiguity but that is not true. For example, the following expression does not specify parameter types:

```
s.setX += o.update;
```

If setX and/or update are overloaded, += operator is applied to all possible combinations of setX and update. Suppose that there are setX(int), setX(String), update(int), and update(String). += operator adds update(int) to setX(int), update(String) to setX(String). If there is update(Object[]), it is added to both setX(int) and setX(String) since it is generic. It is possible to introduce additional syntax for selecting method slots by parameters, but the syntax will be more complicated. Listing 7 is the algorithm in Scheme for picking up and binding two method slots by operators.

Since a language supporting the aspect paradigm must provide a way to retrieve runtime context, for example, AspectJ provides pointcut designators and reflection API for that purpose, DominoJ provides three keywords to retrieve the information about the caller at runtime in the default closure of a method slot. The owner object and the default closure of the method slot at the left-hand side of an operator can be got by using the keywords in the default closure of the method slot at the right-hand side. Unlike AspectJ, which extends the set of pointcut designators available in the language, DominoJ extends the set of special variables such as this and super. In DominoJ a call to the method slot can be regarded as a sequence of method slot calls among objects since a method slot may contain closures calling other method slots. When a method slot is

Listing 7. The algorithm of binding method slots

```
1  ; bind methodslots by operators
2  (define (bind-methodslots operator l_object l_slotname r_object r_slotname)
3    (let ((l_methodslots (get-fields l_object l_slotname))
4          (r_methodslots (get-fields r_object r_slotname)))
5      (for-each
6        (lambda (l_methodslot)
7          (for-each
8            (lambda (r_methodslot)
9              (if (or (same-type? l_methodslot r_methodslot)
10                     (generic-type? l_methodslot r_methodslot))
11               (cond ((equal? operator "=")
12                      (assign-closure l_methodslot r_object r_slotname))
13                     ((equal? operator "^=")
14                      (insert-closure l_methodslot r_object r_slotname))
15                     ((equal? operator "+=")
16                      (append-closure l_methodslot r_object r_slotname))
17                     ((equal? operator "-=")
18                      (remove-closure l_methodslot r_object r_slotname)))))
19           r_methodslots))
20       l_methodslots)))
```

explicitly called by an expression in a certain default closure, the method slots bound to it by operators are implicitly called by DominoJ. Programmers can get the preceding objects in the call sequence. In the default closure, i.e. the body of method slot declaration, the caller object can be got by the keyword $caller. It refers to the object where we start the call sequence by the expression. The predecessor object, in other words, the object owning the preceding method slot in the call sequence, can also be got by the keyword $predecessor. It refers to the object owning the closure calling the current method slot whether explicitly or implicitly. Taking the example of Figure 2, suppose that we have a statement calling s.setX in the default closure of the method slot test in another class Client:

```
public class Client {
  public void test(Shape s) {
    s.setX(10);
  }
}
```

If test on an object instance of this class, for example c, is executed, the relationship between the objects c, s, and o can be described as shown in Figure 3. Note that calling other method slots explicitly by statements in the default closure of test, setX, or update will start separate call sequences. In Figure 3, using $caller in the default closure of setX and update both returns the object c since there is only one caller in a call sequence. However, the predecessor objects of s and o are different. Using $predecessor in the default closure of setX returns the object c, but using $predecessor in the default closure of update returns the object s. Note that both the apparent types of $caller and $predecessor are Object because the caller and the predecessor are determined at runtime. If the current method slot is called in a static method slot, $caller or $predecessor will return the class object properly. The special method call proceed in AspectJ is introduced in DominoJ

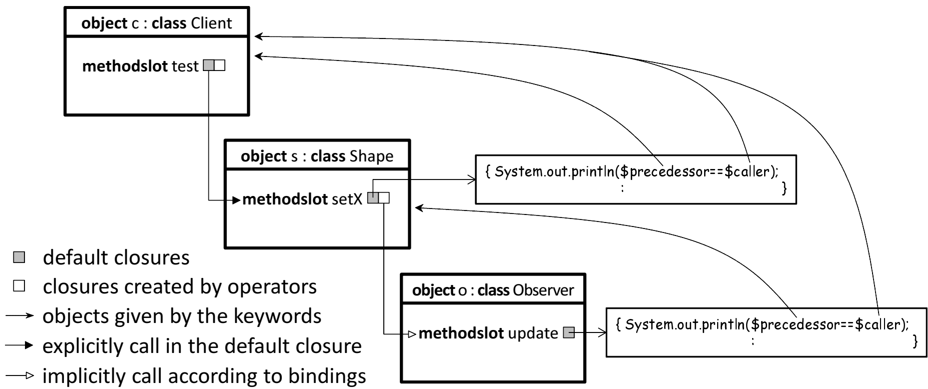


Fig. 3. The keywords $caller and $predecessor

as well. The keyword proceed can be used to call the default closure of the preceding method slot. In Figure 3, calling proceed in the default closure of update on o will execute the default closure of setX on s since s.setX is the preceding method slot of o.update. If there is no preceding method slot for the current one, calling proceed will raise an exception.

4 Evaluation

To show the feasibility of DominoJ and measure the overheads caused by method slots, we implemented a prototype compiler[3] of DominoJ built on top of JastAddJ [28]. The source code in DominoJ can be compiled into Java bytecode and run by Java virtual machine. In the following microbenchmark, the standard library is directly used without recompilation due to the performance concern. All methods in the standard library can be called as method slots which have only the default closure, but cannot be modified by the operators.

4.1 The Implementation

The DominoJ compiler is a source-to-source compiler which translates DominoJ code to plain Java code and then compiles it into Java bytecode. However, implementing the compiler is not easy since closures are not supported by the current Java version (Java 7). In the DominoJ compiler we use the well-known means of the Java language such as inner classes to represent the closures.

Closure Representations in Java. To emulate closures in Java, a naive implementation is using Java reflection. The compiler could generate the code to

[3] The prototype compiler of DominoJ is available from the project webpage: `http://www.csg.ci.i.u-tokyo.ac.jp/projects/dominoj/`

record the target objects and the method names, and use the reflection API to invoke the methods at runtime. For example, adding a closure calling o.update to s.setX could be represented as adding a pair (o, "update"), which an object instance of the class Pair<Object, String>, to the array for s.setX. When s.setX is called, all the pair stored in the array will be iterated and the methods such as o.update can be invoked by the reflection API. It is not surprising that the overheads are not small. Another idea is to define an interface like Callable then a closure can be represented by an object instance of a class implementing the interface. This class is generated by the compiler for every closure. Such an object can be stored in the array for a method slot, and the method inherited from the interface, which contains the method call such as o.update, can be called when the object is iterated.

The DominoJ Compiler. The performance of DominoJ code is determined by how the closures are represented and executed at runtime. Using Java reflection is a naive solution, but the overheads are not negligible. Suppose that we have a method slot setX in DominoJ:

```
public class Shape {
    :
  public void setX(int nx) {
      :  // the default closure
  }
}
```

then the compiler will generate the following Java code in Shape: an array field setX$slot and a method setX for iterating the elements in the array setX$slot. In other words, calling a method slot in DominoJ is translated to calling a method in Java to iterate and invoke the elements in an array as follows:

```
// Java code generated by the compiler
public void setX(int nx) {
  Iterator iter = setX$slot.iterator();
  while(iter.hasNext()) {
      :  // invoke a method
  }
}
```

If we use the reflection API to invoke the methods in the iteration, the array setX$slot must store the target objects and the method names for invoking them:

```
// Java code generated by the compiler
public class Shape {
  public ArrayList<Pair<Object, String>> setX$slot
    = new ArrayList<Pair<Object, String>>();
    :
}
```

where each element in the array setX$slot holds the target object and the method name. Furthermore, the default closure of setX in DominoJ is translated into a method setX$impl in Java, which contains the statements in the default closure. When an object of Shape, for example s, is instantiated, a pair (this, "setX$impl") is appended to the array setX$slot by default. Suppose that we have another method slot update, the parameter types of which is the same as setX:

```
public class Observer {
    :
  public void update(int i) { ... }
}
```

Then the following binding:

```
s.setX += o.update;
```

where o is an object of Observer, is translated into:

```
// Java code generated by the compiler
s.setX$slot.add(new Pair<Object, String>(o, "update"));
```

When s.setX is called, the pairs (this, "setX$impl") and (o, "update") will be got in order. Here we show the code of setX again for demonstrating how to invoke the methods using the reflection API:

```
// Java code generated by the compiler
public void setX(int nx) {
  Class[] pars = new Class[1];
  pars[0] = Integer.TYPE;
  Object[] args = new Object[1];
  args[0] = nx;
  Iterator<Pair<Object, String>> iter = setX$slot.iterator();
  while(iter.hasNext()) {
    Pair<Object, String> pair = iter.next();
    Object obj = pair.getFirst();
    String mname = pair.getSecond();
    Class c = obj.getClass();
    Method m = c.getMethod(mname, pars);
    m.invoke(o, args);
  }
}
```

where the Class array pars is used to specify the parameter types for finding the correct method, (int) in this example, since there may be several overloaded methods. The Object array args, which contains the arguments given to setX. In this example the only argument nx, an int, is autoboxed in an Integer instance and put into args. Obviously the cost of finding and invoking a method using the

reflection API is not low. A possible improvement is storing Method instances instead of the method names, so that we can avoid spending time on finding the Method instance when a method slot is called. However, the cost of invoking a Method is still quite high.

The idea used in our prototype compiler is using an interface to simulate the function closure in JavaScript:

```
// Java code used by the compiler
public interface Closure {
  public Object exec(Object[] args);
}
```

Then for each method slot the compiler can declare a field, which is an anonymous class implementing Closure. For example, the field update$closure is declared in Observer for calling update:

```
// Java code generated by the compiler
public class Observer {
    :
  public Closure update$closure = new Closure() {
    public Object exec(Object[] args) {
      this.update((Integer)args[0]);
      return null;
    }
  }
}
```

Note that the individual element in the array args, the arguments to exec, is typecast properly before giving update. If update is a generic method slot, in other words the only parameter of which is Object[], the array args will be directly given to update:

```
this.update(args);
```

then in the default closure of update programmers need to check the type of each element in the array using instanceof and typecast them if it is necessary. Furthermore, in this example we simply return null in exec since the return type of setX is void. The array for the method slot setX, setX$slot, is now an array of Closure rather than an array of the pair (Object, String):

```
// Java code generated by the compiler
public class Shape {
  public ArrayList<Closure> setX$slot = new ArrayList<Closure>();
    :
}
```

The binding statement we discussed above is now translated into:

```
s.setX$slot.add(o.update$closure);
```

Similarly, a field setX$impl$closure for calling the method setX$impl, which contains the statements in the default closure of setX, is declated in Shape as well:

```
// Java code generated by the compiler
public class Shape {
    :
  public Closure setX$impl$closure = new Closure() {
    public Object exec(Object[] args) {
      this.setX$impl((Integer)args[0]);
      return null;
    }
  }
}
```

In the constructor of Shape the following line is added for appending setX$impl$closure to setX$slot by default:

```
// Java code generated by the compiler
this.setX$slot.add(this.setX$impl$closure);
```

When the method slot setX is called, all Closure instances in the array are iterated and their exec methods are called with args, the Object array containing the arguments given to the method slot setX, in this example only nx:

```
// Java code generated by the compiler
public void setX(int nx) {
  Object[] args = new Object[1];
  args[0] = nx;
  Iterator<Closure> iter = setX$slot.iterator();
  while(iter.hasNext()) {
    Closure c = iter.next();
    c.exec(args);
  }
}
```

The iteration is similar to the reflection version, but the code for invoking a method using the reflection API is replaced with a call to the exec method in Closure. In other words, we need more memory to hold the Closure instances, but the overheads of method slots can be reduced to the cost of calling the exec method.

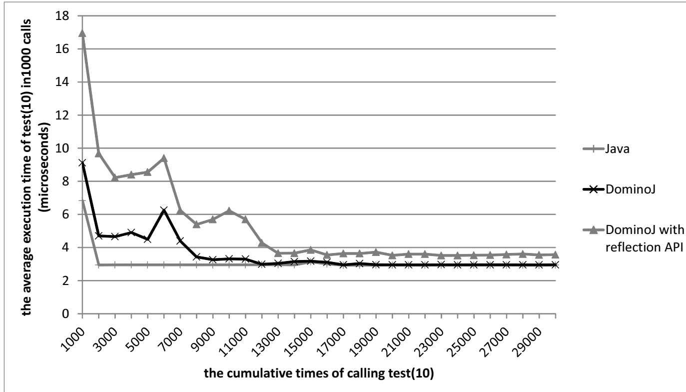


Fig. 4. The average time of continuously calling a method in Java and DominoJ

4.2 Microbenchmark

In order to measure the overheads of method slots, we executed a simple program and compared the average time per method call in DominoJ and in plain Java. The method we measure is named test, which calculates $sin(\pi/6)$ by expanding Taylor series up to 100th order a number of times according to the argument as shown below:

```
private double x = 3.141592653589793 / 6;
private double result = 0;
public void test(int count) {
  for(int i=0; i<count; i++) {
    double sum = x;
    double n = x;
    double d = 1;
    for(int j=3; j<100; j+=2) {
        n *= - x*x;
        d *= (j-1)*j;
        sum += n/d;
    }
    result = sum;
  }
}
```

Figure 4 shows the results of continuously calling test(10) and calculating the average execution time of calling test(10) every 1000 times of calls until the total amount of calls reaches 30000. For example, the first values we calculate are the average of execution time of 1st 1000th calls in Java and in DominoJ, and the second values are the ones of 1001st 2000th.

The program was compiled by our prototype compiler and run on the JVM of OpenJDK 1.7.0_25 and Intel Core i7 (2.67GHz, 4 cores) with 8GB memory. The result of the naive implementation using the reflection API we mentioned in Section 4.1 is also shown for comparison. After the optimization is sufficiently applied by the JIT compiler, the overhead is negligible (2955ns against 2932ns) although it is initially about 34% (9124ns against 6833ns). On the other hand, the overheads of the reflection version is about 20% (3516ns against 2932ns) after the optimization.

To measure the performance of an operation on method slots such as assigning a closure to a method slot using `=` operator, we repeated the operation and calculated the average time as follows:

```
long start = System.nanoTime();
for(int j=0; j<1000; j++) {
    s.setX = o.update;
}
long estimated = System.nanoTime() - start;
System.out.println(estimated/1000);
```

We also measured other operations by adding one more statement, which uses the other operators such as `+=` operator after the assignment:

```
  :
for(int j=0; j<1000; j++) {
    s.setX = o.update;
    s.setX += o.update;
}
  :
```

Figure 5 shows the result of running such programs 100 times. According to the average time of the operations in the four programs, we can calculate the time of the four operations: the `=` operation takes 427ns, the `^=` operation takes 483ns, the `+=` operation takes 275ns, and the `-=` operation takes 726ns. The `-=` operation might be even slower when the number of closures in the method slot is large since it takes time to check every closure in the array. It is reasonable that the `+=` operation is the fastest one since it simply appends to the array, while the `=` operation have to clear the array and the `^=` operation inserts to the beginning of the array; the performance is relevant to how the method slots are implemented. Finding a more efficient technique to implement method slots is included in our future work. For example, using other structures instead of ArrayList to store the closures or using the new JVM instruction invokedynamic to emulate the closures might be possible solutions to improve the performance of DominoJ code.

4.3 Method Slots and Design Patterns

Method slots extend the method paradigm to support the event-handler paradigm and the aspect paradigm, while still preserving the original behavior in the method

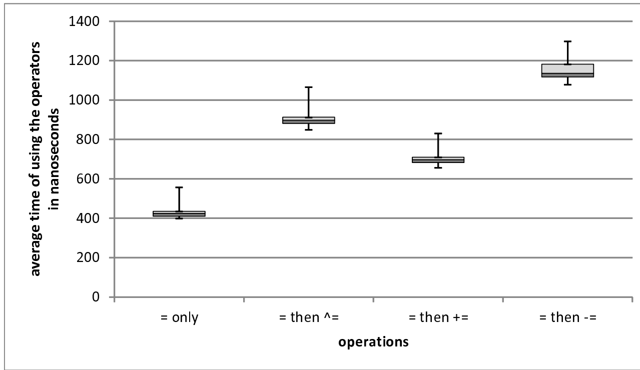


Fig. 5. The average time of using the operators for method slots

paradigm. In DominoJ, if the operators for method slots are not used, the code works as in plain Java. In other words, a Java method can be regarded as a method slot that has only the default closure.

We could regard the inheritance in object-oriented programming as an event mechanism with default bindings. A method declaration in the superclass is an event declaration, and its implementation is the handler bound to the event by default. If the method is overridden in a subclass, the overriding implementation automatically replaces the overridden one and becomes the only handler for the event. In other words, the call to a method on an object is an event, and the method implementation selected by the polymorphism is the handler. The binding from the handler to the event in the inheritance is a one-to-one relation and predefined. Method slots extend the default binding in object-oriented programming to allow the binding of more than one handler to an event.

We have also analyzed how method slots can be applied to "GoF" design patterns [7], and classify the patterns into four groups as shown in Table 2. Furthermore, we implemented the sample code in the GoF book in Java and DominoJ, and compared them with respect to the four modularity criteria borrowed from [10], and a new criterion named noninheritance, which means that method slots can be used as an alternative to the inheritance solution or not. This might remind readers of the mixin. As an alternative to the inheritance both the mixin and method slots allow to execute an implementation in another class or object for a method call at runtime. However, in several mixin mechanisms both fields and methods are included, but a binding between method slots do not involve fields. The comparison is shown in Table 3, where the number of lines of code is listed as well. Note that for the patterns in group III we ignore the comparision on the number of lines of code since in group III method slots do not act a major role as in group I and group II. In this table the locality means the code of defining the relation can be gathered up, the reusability means the pattern code can be abstracted and thus reusable, the composition means the code do not get complicated when applying multiple relationships to the same class, and the unpluggability means it is easy to apply or remove

the pattern. The operators for method slots can be used to cause the execution of a method slot on another object when a specified method slot is called. In general such mixin behavior helps to gather up similar implementations in a class and can be an alternative to the polymorphism. Furthermore, the pattern code for propagating events can be expressed by the bindings, which can be gathered up in one place for the locality. For several patterns such as the Chain of Responsibility pattern, the event implementation is almost eliminated and thus the code tangling caused by the composition can be avoided. If the pattern code can be totally eliminated, reusing it is quite easy since there is no need to implement the pattern every time. It is also possible to make the code easy to plug or unplug for several patterns such as the Proxy pattern since the pattern is applied by a binding rather than passing a different object. However, unlike the polymorphism the switch between different implementations must be manually managed. As to the numbers of lines of code in Java and in DominoJ, basically there are no significant difference since the explicit triggers for events are removed but the bindings for describing the event propagations are added. However, for several patterns such as the Observer pattern the pattern code of which is totally eliminated. On the other hand, using the mixin behavior in several patterns such as the Factory Method pattern takes additional lines of bindings to switch the implementations. Below we discuss the four groups by showing concrete examples.

Table 2. Method slots can be applied to design patterns

	Pattern Name	**Description and Consequences**
I	Adapter, Chain of Responsibility, Composite, Decorator, Facade, Mediator, Observer, Proxy	*Implicitly propagate events among objects by the bindings.* *GOOD: The bindings can be gathered up in one place.* *BAD: The method slots which handle the same event must share the same type.*
II	Abstract Factory, Bridge, Builder, Factory Method, State, Strategy, Template Method, Visitor	*Change class behavior at runtime without inheritance.* *GOOD: A solution to avoid multiple inheritance.* *BAD: Unlike the polymorphism the switch between implementations have to be manually managed.*
III	Command, Flyweight, Interpreter, Iterator, Prototype	*Replace inheritance part in the logic.* *GOOD: Provide an alternative for the inheritance part.* *BAD: Not helpful except the inheritance part.*
IV	Memento, Singleton	*Not applicable*

The key idea of the patterns in group I can be considered event propagation—from the outer object to the inner object, or among colleague objects. Using method slots can avoid code scattering caused by the pattern code since event implementation is eliminated. Code tangling caused by combining multiple patterns can be eased as well. The following example is an example of the Chain of Responsibility pattern. In a graphical user interface library a widget such as a button may need a hotkey for showing help. When users are confused with the label of the button, they can press the `F1` key to get a pop-up description, which explains the meaning in detail. To implement the help event in Java, the Chain of Responsibility pattern can be used in `Widget`, the base class for all widgets, as shown in Listing 8. Here we assume that the method `handleHelp` will be called when users press the `F1` key on a widget object. Every subclass of

Table 3. The benefit of applying DominoJ to design patterns

		Modularity Properties					#Lines of Sample Code	
	Pattern Name	Locality	Reusability	Composition	Unpluggability	Non-inheritance	in Java	in DominoJ
I	Adapter	✓					51	48
I	Chain of Responsibility	✓	✓	✓	✓	✓⋆	38	28
I	Composite		✓	✓	✓	✓⋆	41	16
I	Decorator	✓		✓	✓		26	20
I	Facade	✓⋆					34	53
I	Mediator	✓			✓		68	49
I	Observer	✓	✓	✓	✓	✓⋆	71	32
I	Proxy				✓	✓⋆	47	61
II	Abstract Factory	✓⋆				✓⋆	41	58
II	Bridge	✓⋆				✓⋆	58	64
II	Builder	✓⋆				✓⋆	55	69
II	Factory Method	✓⋆				✓⋆	67	97
II	State	✓					66	69
II	Strategy			✓	✓	✓⋆	36	28
II	Template Method	✓				✓⋆	31	45
II	Visitor	✓		✓	✓		63	69
III	Command					✓⋆	Ignored	
III	Flyweight					✓⋆		
III	Interpreter					✓⋆		
III	Iterator					✓⋆		
III	Prototype					✓⋆		
IV	Memento	Same implementation for Java and DominoJ						
IV	Singleton							

The ✓ mark means that DominoJ has better modularity than Java when implementing the pattern.
The ⋆ mark means that AspectJ does not provide such modularity when implementing the pattern, while DominoJ does.

the Widget class should override the handleHelp method to implement its own behavior for the help event, and return a boolean value to indicate whether the help event is handled or not. In the Widget class a default implementation is given: propagating the help event to the successor in the chain of responsibility. The successor is kept as a private field and set to its container in the constructor as shown in Line 2-5. If no successor is set, false is returned. When a subclass of Widget such as Button class overrides the handleHelp method, it must explicitly call super.handleHelp for executing the default implementation to propagate the help event to its successor. In DominoJ, the operator += can be used to describe such behavior as shown in Listing 9. Note that in Line 10 the keyword $retval is used to check if the help event is handled by the predecessor, and the explicit call super.handleHelp is removed from all subclasses. It makes the code clear, especially when there are several chain of responsibility for different events in the Widget class. Using DominoJ can avoid the tangling caused by pattern code.

Method slots can also be used to improve the transparency to clients. In a class-based object-oriented language such as Java, it is not allowed to change the class membership of objects as discussed in [4]. Suppose that two classes Student and Employee are given to model the students and the employees in a university. If now a student has graduated and employed by the university, we cannot continue using the original Student object. We have to create a new Employee object according to the original Student object and update all references to the object in clients. A solution is using method slots to implement the Proxy pattern for the Student example. In the Proxy pattern usually the clients are aware of the existence of the proxy object. For example, in order to control the access to Student, giving a proxy class Employee, which owns a reference to its original Student object, then the clients have to use the proxy object instead of the original Student object. In DominoJ the behavior of a Student object such as getInfo can be replaced if it is public:

```
s.getInfo = e.getInfo;
```

Listing 8. The Chain of Responsibility pattern example in Java

```
1  public class Widget {
2    private Widget successor = null;
3    public Widget(Widget container) {
4      successor = container;
5    }
6    public boolean handleHelp() {
7      if(successor == null)    return false;
8      return successor.handleHelp();
9    }
10       :
11 }
12 public class Button extends Widget {
13   public boolean handleHelp() {
14   : //return true if it can offer help,otherwise return super.handleHelp()
15   }
16 }
```

Listing 9. The Chain of Responsibility pattern example in DominoJ

```
1  public class Widget {
2    public Widget(Widget container) {
3      this.handleHelp += container.handleHelp;
4    }
5    public boolean handleHelp();
6        :
7  }
8  public class Button extends Widget {
9    public boolean handleHelp() {
10       if($retval)    return true;
11       :  // return true if it is handled here, otherwise return false
12   }
13 }
```

where s is a Student object and e is its proxy, an Employee object. Then the clients of s may continue using the reference to s. When s.getInfo is called, the method slot getInfo on its proxy object will be executed for access control. In other words, it is possible to make the clients unaware of plugging or unplugging the proxy.

The patterns in group II use the inheritance to alter the class behavior at runtime. Different implementation for a method slot call can be added to the method slot instead of overriding in subclasses. In that sense, method slots can be used as an alternative to the polymorphism. Although method slots are not perfect replacement for the inheritance, it is convenient in particular when programmers are forced to choose between two superclasses due to single inheritance limitation. For example, Listing 10 shows an example of the Template Method pattern in Java. By taking advantage of inheritance, the drawing border step in the class View can be deferred to its subclass FancyView by overriding the method drawBorder. However, unlike mixin or multiple inheritance, in the subclass FancyView we cannot reuse the implementation of other classes due to the single inheritance limitation in Java. For example, the implementation of drawBorder in FancyView may be the same as the one in another class FancyPrint,

Listing 10. The Template Method pattern example in Java

```
1  public class View {
2    public void display() {
3      drawBorder();
4      drawContent();
5    }
6    public void drawBorder() {
7      System.out.println("View: drawBorder");
8    }
9    public void drawContent() {
10     System.out.println("View: drawContent");
11   }
12 }
13 public class FancyView extends View {
14   public void drawBorder() {
15     System.out.println("Fancy: drawBorder");
16   }
17 }
```

which is neither a subclass of View nor a subclass of FancyView. In this case we cannot extract the common part of FancyView and FancyPrint into a new class Fancy. In DominoJ such mixin behavior is possible by using the operator =. As shown in Listing 11 we move the drawBorder implementation to a new class Fancy and let FancyView own a reference to a Fancy object. Then in the constructor of FancyView we can forward the call to its method slot drawBorder to the one in the Fancy object it refers (Line 22). With DominoJ a subclass can still benefit from another class by the binding as using the mixin. It helps to modularize the code when we want to extract parts of the implementation in the subclass. Programmers can decide to use mixin or inheritance for a feature depending on the design.

Another example we want to show here is the State pattern, which allows an object to alter its behavior by switching between the state objects. Using DominoJ the state transitions can be modularized in another class as using AspectJ [10]. Suppose that we have three state classes for the Queue class: QueueEmpty, QueueNormal, and QueueFull. In Java the state transition code scatters across the state classes, for example the transition from QueueEmpty to QueueNormal is checked and performed in the insert method of QueueEmpty class as shown in Listing 12. In DominoJ all transitions can be gathered up in another class UpdateQueueState as shown in Listing 13. The class UpdateQueueState keeps all the state objects (Line 2-4) and manages the transitions such as emptyToNormal (Line 6-11). For example, the transition emptyToNormal is performed after the method slot insert on the object empty is executed as shown in Line 16. Note that the method slots emptyToNormal and insert share the same type.

The patterns classified under group III also use the inheritance as a part of their pattern code, so programmers may use method slots or not depending on the situation. For example, the intent of the Command pattern is wrapping the requests in objects in order to pass around clients, and inheritance is used for overriding the behavior of a request. Suppose that we want to implement a document editor, which allows users to open a document, edit its content, and

Listing 11. The Template Method pattern example in DominoJ

```
1  public class View {
2    public void display() {
3      drawBorder();
4      drawContent();
5    }
6    public void drawBorder() {
7      System.out.println("View: drawBorder");
8    }
9    public void drawContent() {
10     System.out.println("View: drawContent");
11   }
12 }
13 public class Fancy {
14   public void drawBorder() {
15     System.out.println("Fancy: drawBorder");
16   }
17 }
18 public class FancyView extends View {
19   Fancy fancy;
20   public FancyView() {
21     fancy = new Fancy();
22     this.drawBorder = fancy.drawBorder;
23   }
24 }
```

copy a paragraph. First we declare an abstract class Command, which has a method slot execute, to model the commands supported in the editor:

```
public abstract class Command {
    :
  public void execute();
}
```

Then we can implement the individual commands such as OpenCommand and CopyCommand by extending the Command class. In the subclasses we can declare necessary parameters and override execute to define the behavior for individual commands. For example, the implementation of OpenCommand looks like this:

```
public class OpenCommand extends Command {
  private File file = null;
    :
  public void execute() {
    file = getFileFromUser();
  }
}
```

Here, the user has to select a file and then the path in the field file will be stored when its execute is called. By creating the command objects, the requests from users can be wrapped and passed to other UI components. The functionalities such as undo and redo can also be implemented easily. In group III the inheritance is not the core of the pattern code, but helps the implementation.

Listing 12. The State pattern example in Java

```
1  public class Queue {
2    private QueueState state = new QueueEmpty();
3    public void setState(QueueState s) {
4      state = s;
5    };
6    public boolean insert(Object o) {
7      return state.insert(this, o);
8    }
9      :
10 }
11 public class QueueState {
12   public boolean insert(Queue q, Object o) {
13     return false;
14   };
15     :
16 }
17 public class QueueEmpty extends QueueState {
18   public boolean insert(Queue q, Object o) {
19     QueueNormal nextState = new QueueNormal();
20     q.setState(nextState);
21     return nextState.insert(q, o);
22   }
23     :
24 }
```

As the example of the Template Method pattern shown above in group II, the inheritance can be replaced with the mixin by using method slots. Again, using the mixin is not always a good choice and it depends on programmers' design decision.

As to the patterns in group IV, DominoJ is not helpful in dealing with object creation as what AspectJ does in [10]. The reason is that DominoJ does not support intertype declaration and cannot stop the object creation. Further details of this analysis is available in [34].

4.4 The Event-Handler Paradigm

There are three important metrics to evaluate an event mechanism. First, the amount of explicit triggers in a program depends on whether the events can be implicit or not. Second, if dynamic binding is not provided, it is not possible to change the handler at runtime. Third, event composition helps the abstraction though it is not absolutely necessary. In an event mechanism the three properties are determined by how the bindings between the event and the handler are presented.

To evaluate how DominoJ works for the event-handler paradigm, first we analyze the bindings between the event and the handler in a typical event mechanism like EScala, and compare them with DominoJ. In languages directly supporting the event-handler paradigm, events are usually introduced as fields, which are separate from methods. In order to associate fields with methods, there are three types of binding between events (fields) and handlers (methods). The ways used for each type of binding are usually different in an event mechanism, and also

Listing 13. The State pattern example in DominoJ

```
1  public class UpdateQueueState {
2    private QueueEmpty empty = new QueueEmpty();
3    private QueueNormal normal = new QueueNormal();
4    private QueueFull full = new QueueFull();
5    private Queue queue = null;
6    public boolean emptyToNormal(Object o) {
7      normal.insert(o);
8      queue.setState(normal);
9      return $retval;
10   }
11     :
12   public void setup(Queue q) {
13     queue = q;
14     queue.setState(empty);
15     empty.insert += this.emptyToNormal;
16       :
17   }
18 }
```

different between event mechanisms. Table 4 shows the ways provided by EScala. The corresponding DominoJ syntax for the three types of binding is also listed, but actually there is only slot-to-slot binding in DominoJ since only method slots are involved in the event-handler paradigm. Every method slot can play an event role and a handler role at the same time. Listing 14 shows how to use DominoJ for the event-handler paradigm for the shape example mentioned in Section 2. Below we will discuss what the three types of binding are, and explain how DominoJ provides the same advantages with the simplified model.

The event-to-handler binding is the most trivial one since it means what action reacts to a noteworthy change. Whether supporting the event-handler paradigm by languages or not, in general the event-to-handler binding is dynamic and provided in a clear manner. For example, in the Observer pattern an observer object can call a method on the subject to register itself; in C# and EScala, += operator and -= operator are used to bind/unbind a method to a special field named event. In addition to the two operators, DominoJ provides ^= operator and = operator to make it easier to manipulate the array of handlers. In C# and EScala, the handlers for an event can be only appended sequentially and removed individually, but in DominoJ, programmers can use = operator to empty the array directly without deducing the state at runtime. Using ^= operator along with += operator also makes design intentions more clear since a closure can be inserted at the beginning without popping and pushing back.

The second one is the event-to-event binding that enables event composition and is not always necessary but greatly improves the abstraction. In a modern event mechanism, event composition should be supported. EScala allows programmers to define such higher-level events to make code more readable. An event-to-event binding can be simulated by an event-to-handler binding and a handler-to-event binding, but it is annoying and error-prone. In DominoJ, it is also possible to define a higher-level event by declaring a method slot without a default closure. Then operators += and ^= can be used to attach other events like what the operator || in EScala does. Other operators in EScala such as &&

Table 4. The roles and bindings of the event-handler paradigm in EScala and DominoJ

<table>
<tr><th></th><th>Type</th><th>EScala</th><th>DominoJ</th></tr>
<tr><td rowspan="2">role</td><td>Event</td><td>field (evt)</td><td rowspan="2">method slot</td></tr>
<tr><td>Handler</td><td>method</td></tr>
<tr><td rowspan="7">binding</td><td rowspan="2">Event-to-Handler</td><td>+=</td><td>+=</td></tr>
<tr><td>-=</td><td>-=</td></tr>
<tr><td rowspan="2">Event-to-Event</td><td>||</td><td>+=, ^=</td></tr>
<tr><td>&&
\
filter
map
empty
any</td><td>use Java expression in the default closure of method slots</td></tr>
<tr><td rowspan="3">Handler-to-Event</td><td>afterExec</td><td>+=</td></tr>
<tr><td>beforeExec</td><td>^=</td></tr>
<tr><td>imperative</td><td>explicit trigger is possible</td></tr>
</table>

and map are not provided in DominoJ, but the same logic can be represented by statements in another handlers and attached by += operator. For example, in Listing 1 we can declare a new event adjusted that checks if left and top are the same as the arguments given to setPosition using the operator && in EScala:

```
evt adjusted[Unit] = afterExec(setPosition)
                      && ((left,top) != _._1)
adjusted += onAdjusted
```

where `_._1` refers to the arguments given to setPosition and onAdjusted is the reaction. In DominoJ, we can declare a higher-level event adjusted and perform the check in another method slot checkAdjusted:

```
public void adjusted(int x, int y);
public void checkAdjusted(int x, int y) {
  if(!(x==left && y==top))  adjusted(x, y);
}
```

and then bind them as follows:

```
setPosition += checkAdjusted;
adjusted += onAdjusted;
```

Although the expression in DominoJ is not rich and declarative as in EScala, they can be used to express the same logic. In addition, the event-to-event binding in EScala is static, so that the definition of a higher-level event in EScala cannot be changed at runtime. On the other hand, it is possible in DominoJ since the slot-to-slot binding is totally dynamic.

The last one is handler-to-event binding, which is also called an event trigger or an event definition. It decides whether an event trigger can be implicit or not. In the Observer pattern and C#, an event must be triggered explicitly, so

Listing 14. Using DominoJ for the event-handler paradigm

```
1  public class Display {
2    public void refresh(int x, int y) {
3      System.out.println("display is refreshed.");
4    }
5  }
6  public class Shape {
7    private int left = 0; private int top = 0;
8    public void setPosition(int x, int y) {
9      left = x; top = y;
10   }
11   public Shape(Display d) {
12     this.setPosition += d.refresh;
13   }
14 }
15 public class Test {
16   public static void main(String[] args) {
17     Display d = new Display();
18     Shape s = new Shape(d);
19     s.setPosition(0, 0);
20   }
21 }
```

that the trigger code is scattering and tangling. EScala provides two implicit ways and an explicit way: after the execution of a method, before the execution of a method, or triggering an event imperatively. In DominoJ, an event can be triggered either implicitly or explicitly. A method slot can not only follow the call to another method slot but also be imperatively called. More precisely, there is no clear distinction between the two triggering ways. In EScala, afterExec and beforeExec are provided for statically binding an event to the execution of a method while DominoJ provides += operator and -= operator for dynamically binding a method slot to the execution of another method slot. This sounds like that a method slot has two predefined EScala-like events for the default closure, but it is not correct. In DominoJ's model the only event is the call to a method slot, and the default closure is also a handler like the other closures calling other method slots. This feature makes the code more flexible since the execution order of all handlers can be taken into account together. As to the encapsulation, in EScala the visibility of explicit events follows its modifiers, and the implicit events are only visible within the object unless the methods they depend on are observable. On the other hand, the encapsulation in DominoJ relies on the visibility of method slots. The design is simpler but limits the usage because a public method slot is always visible as an event to other objects.

There is one more important difference between EScala and DominoJ. In DominoJ, a higher-level event can be declared or not according to programmers' design decision. In order to explain the difference, we use a tree graph to represent the execution order in the shape example by regarding setPosition as the root. As shown in Figure 6, we use rectangles, circles, and rounded rectangles to represent methods, events, and method slots, respectively. When a node is called, the children bound by beforeExec or ^= must be executed first, followed by the node itself and the children bound by afterExec or +=. Figure 6 (a) is the execution order of Listing 1, and Figure 6 (b) is the one of Listing 14. In the

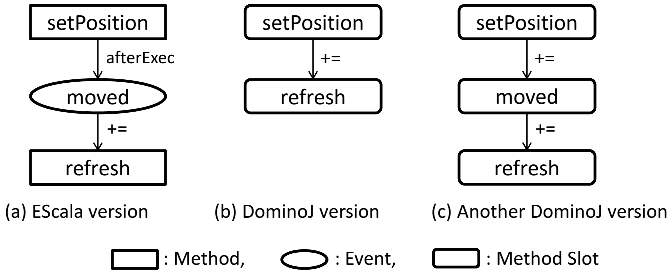


Fig. 6. The execution order of the shape example in EScala and DominoJ

DominoJ version, the event moved is eliminated and its child refresh is bound to setPosition directly since we do not need additional events in such simple case. DominoJ is easier and simpler to apply the event-handler paradigm when events are not complicated but used everywhere. In EScala, events must be created since methods cannot be bound to each other directly. However, such events are still necessary if we want to keep the abstraction. In that case, method slots can be used as the events in EScala by declaring them without a default closure. For example, the event moved in Line 11 of Listing 1 can be translated into the following statements:

```
public void moved();
setPosition += moved;
```

Figure 6 (c) is another DominoJ version, which has the higher-level event as the EScala version. In DominoJ, programmers can choose between the simplified one and the original one depending on the situation.

Note that the number of lines of Listing 14 is one line longer than Listing 1 because the syntax of Scala looks more compact than Java. In Java the constructor and the fields used inside a class must be declared explicitly while they are omitted in Scala. In Listing 14 the constructor takes two more lines than Listing 1. If we do not take this into account, the EScala version is one line longer than the DominoJ version due to additional event declaration.

The line of code can also be analyzed according to Table 4. With regard to the roles, additional event declarations are necessary in EScala while they are combined into one declaration in DominoJ as we discussed above. For the event-to-handler binding, both the operators provided by EScala and DominoJ take one line. For the event-to-event binding, the operators provided by EScala can be written in the same line, but in DominoJ += operator and ^= operator cannot be merged into one line. In that case the code in DominoJ is longer than the EScala one. For example, a higher-level event changed can be defined by three events resized, moved, and clicked:

```
evt changed[Unit] = resized || moved || clicked
```

but in DominoJ they must be defined as follows:

```
resized += changed;
moved += changed;
clicked += changed;
```

That is why the expression in EScala is richer but complicated. Introducing appropriate syntax sugar to DominoJ to allow to put operators in one line is also possible, but we think it makes the design complicated. However, in this example we can also find passing the event value in EScala takes effort. In EScala, as far as we understand, only a value is kept in an event field. If we want to gather up the arguments x and y given to setPosition, and then pass to moved and changed, we need to declare additional classes such as Point and declare the events with the new type rather than Unit[4]. The additional classes increase the number of lines as well. For the handler-to-event binding, afterExec and beforeExec in EScala can define an anonymous event and share the same line of an event-to-handler binding. To sum up, in DominoJ the event declarations may be eliminated and thus the number of lines of source code can be reduced. On the other hand, the number of code of DominoJ version is longer when translating a complex EScala expression composed of a number of operators since DominoJ has less primitive syntax. DominoJ makes code clear because each method slot has a name explicitly, and each line for binding only defines the relation between two method slots.

4.5 The Aspect Paradigm

DominoJ can be used to express the aspect paradigm as well. In order to discuss language constructs concretely, we compare DominoJ with the most representative aspect-oriented language—AspectJ. The call to a method slot is a join point, and other method slots can be bound to it as advices. Note that aspect-oriented programming is broader as discussed in [14] and not restricted to the AspectJ style, which is the point-advice model. In AspectJ the important features such as around advices, the obliviousness, and intertype declaration that an event mechanism cannot provide are all supported by constructs. In this subsection first we analyze the necessary elements in the point-advice model in order to compare the constructs provided by AspectJ and DominoJ. Then we use DominoJ to rewrite the shape example in Listing 2 and discuss the differences.

Since the purpose of the aspect paradigm is to modularize the crosscutting concerns, we need a method-like construct to contain the code piece, a way to attach the method-like construct to a method execution, and a class-like construct to group the method-like construct. In AspectJ, the class-like construct is the aspect construct, the method-like construct is the advice body, and the way of attaching is defined by the pointcut and advice declaration. In DominoJ, the method slot and the class construct in plain Java are used and only operators

[4] In EScala, declaring events with Unit type means that no data are passed [9].

for method slots are introduced for attaching them. The method slots bound by += operator or ^= operator are similar to after/before advices, respectively. The method slots bound by = operator are similar to around advices and proceed can be used to execute the original method slot. It is expected that DominoJ cannot cover all expression in AspectJ since DominoJ's model is much simpler. For example, in DominoJ intertype declaration and the reflection are not provided. According to the three elements, Table 5 lists the mapping of language constructs in AspectJ and DominoJ.

Table 5. The mapping of language constructs for the aspect paradigm in AspectJ and DominoJ

Construct	AspectJ	DominoJ
grouping	aspect	class
code piece	advice body	method slot body (default closure)
pointcut and advice declaration	after returning and execution	+= and $retval
	before and execution	^=
	around	=
	this	$caller
	target	$predecessor
	args	by parameters

In AspectJ programmers need to understand the special instance model for the aspect construct, but in DominoJ the class construct is reused. Although the instances of the construct for grouping need to be managed manually, there is no need to learn the new model and keywords like issingleton, pertarget, and percflow. In DominoJ programmers can create an instance of the aspect-like class and attach its method slots to specified objects according to the conditions at runtime. If the behavior of issingleton is preferred, programmers can declare all fields including method slots in the aspect-like class as static since static method slots are supported by DominoJ. The shape example of AspectJ in Section 2 can be rewritten by DominoJ as shown in Listing 15. Here the class UpdateDisplay is the aspect-like class. In Line 14, we attach the advice refresh in a static method slot init, so all Shape objects will share the class object of UpdateDisplay. Furthermore, we let init be executed after the constructor of Shape, so that we can avoid explicitly attaching refresh every time a Shape object is created. Moreover, we do not have to modify the constructor of Shape. If we need to count how many times setPosition is called for each Shape and thus pertarget is preferred, we can rewrite the class UpdateDisplay as shown in Listing 16. Every time a Shape object is created, a UpdateDisplay object is created for it implicitly. Note that the object ud will not be garbage-collected since its method slot count is attached to another method slot.

In DominoJ, there is no difference between methods and advices while in AspectJ they are different constructs. Although an advice in AspectJ can be regarded as a method body, it cannot be directly called. If the code of an advice

Listing 15. Using DominoJ as the aspect paradigm

```
1  public class Display {
2    public static void refresh(int x, int y) {
3      System.out.println("display is refreshed.");
4    }
5  }
6  public class Shape {
7    private int left = 0; private int top = 0;
8    public void setPosition(int x, int y) {
9      left = x; top = y;
10   }
11 }
12 public class UpdateDisplay {
13   public static void init() {
14     ((Shape)$predecessor).setPosition += Display.refresh;
15   }
16   static { Shape.constructor += UpdateDisplay.init; }
17 }
18 public class Test {
19   public static void main(String[] args) {
20     Shape s = new Shape();
21     s.setPosition(0, 0);
22   }
23 }
```

Listing 16. Rewrite UpdateDisplay for pertarget

```
1  public class UpdateDisplay {
2    private int total = 0;
3    public void count(int x, int y) {
4      total++;
5    }
6    public static void init() {
7      UpdateDisplay ud = new UpdateDisplay();
8      ((Shape)$predecessor).setPosition += ud.count;
9    }
10   static { Shape.constructor += UpdateDisplay.init; }
11 }
```

is reusable, in AspectJ we must move it to another method but in DominoJ it is not necessary.

The pointcut and advice declaration in AspectJ and DominoJ are similar but not the same. First, what they target at is different. AspectJ is class-based while DominoJ is object-based. In other words, what AspectJ targets at are all object instances of a class and its subclasses but what DominoJ targets at are individual object instances. However, it is possible to emulate the class-based behavior in DominoJ by the code attaching to the constructor of a class as shown in Line 16 of Listing 15. Second, unlike AspectJ that has call and execution pointcut, in DominoJ only execution pointcut is supported. This limits the usage but reduces the complexity. In fact, the relation between advices is quite different in AspectJ and DominoJ. In AspectJ an advice is attached to methods and cannot be directly attached to a specific advice, but in DominoJ a method slot is not only an advice but also a method. For example, if we need another advice for checking the dirty region in Listing 2, we may prepare an aspect CheckDirty containing this advice as shown in Figure 7 (a). However, the advice can only

be attached to setPosition. In DominoJ, the advice can be attached to either setPosition or init as shown in Figure 7 (b).

The behavior of proceed in AspectJ and DominoJ is also a little different. The proceed in DominoJ should be used only along with = operator since it calls the default closure in the preceding method slot rather than the next closure. The root cause of the difference is the join point model: what DominoJ adopts is the point-in-time model while the one AspectJ adopts is the region-in-time model [15]. In other words, in AspectJ the arrays of the three types of advices are separate, but in DominoJ there is only one array. If += operator or ^= operator are used after using = operator to attach a method slot containing proceed, the behavior is not as expected as in AspectJ. Figure 8 shows an example of around advices in AspectJ and DominoJ. In AspectJ, the around advices localCache and memCache are attached to queryData in order. In DominoJ, we can do it similarly:

```
queryData = localCache;
localCache = memCache;
```

then using proceed in memCache and localCache will call the default closure of their preceding method slot, localCache and queryData, respectively. Another difference is that the args pointcut and the wildcard used in call and execution pointcuts in AspectJ are not supported in DominoJ. Method slots are simply matched by their parameters. If the overloading is not taken into account, the operators in DominoJ only select one method slot in one line statement.

As for the number of lines, the two versions are about the same. Comparing them line by line might not make much sense since there is no simple translation between DominoJ and AspectJ.

4.6 Summary of the Coverage

In the previous subsections we have discussed what a language must have for the event-handler paradigm and the aspect paradigm by comparing with EScala and AspectJ, respectively, from the viewpoint of constructs. In this subsection we summarize the significant characteristics of the two paradigms and discuss the support in DominoJ as shown in Table 6. In addition to being used for the event-handler paradigm and the aspect paradigm, DominoJ allows programmers to use both paradigms together.

For the event-handler paradigm, there are three significant properties: implicit events, dynamic binding, and event composition. DominoJ supports them all by method slots and only four operators. Rewriting a complex expression of event composition in EScala is also possible though it takes more lines. Introducing additional syntax may resolve the issue but it also complicates the model. As a result of regarding method slot calls as events, giving an event a different visibility from the method slots it depends on is not supported by DominoJ.

The aspect paradigm of AspectJ has three important features that cannot be provided by the event-handler paradigm: around advices, the obliviousness, and inter-type declaration. In DominoJ what the around advices in AspectJ does

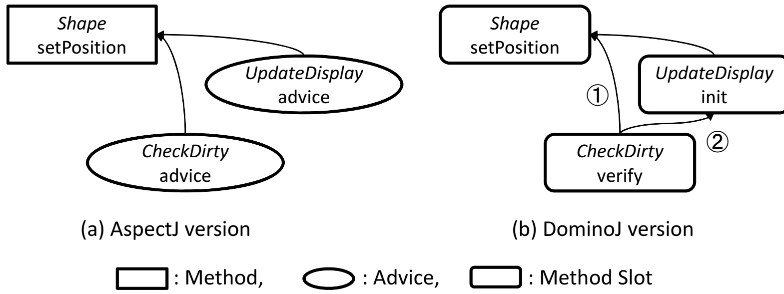


Fig. 7. Adding another advice to the shape example in AspectJ and DominoJ

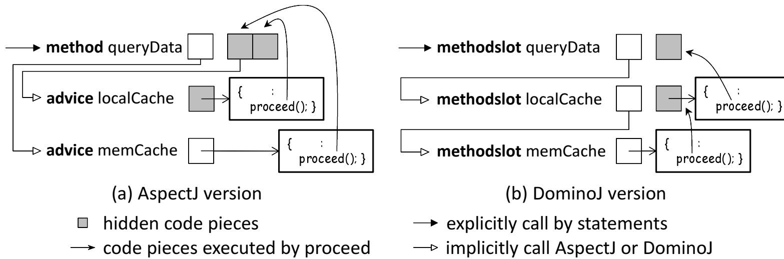


Fig. 8. Calling proceed in AspectJ and DominoJ

Table 6. A summary of the significant characteristics of the two paradigms and the support in DominoJ

the Event-handler paradigm	DominoJ
implicit events	yes
dynamic binding	yes
event composition	yes

the Aspect paradigm	DominoJ
around advices	yes
the obliviousness	yes
inter-type declaration	no

can be archived by assigning a closure calling another method slot using the = operator. DominoJ also supports the obliviousness in AspectJ by using the class construct as the aspect construct and attaching a method slot to a constructor of the target class. In the method slot attached to the constructor, programmers can further attach advices to the method slots at the target class. However, the intertype declaration in AspectJ is not available in DominoJ. A possible solution is introducing a default method slot for undefined fields in a class like Smalltalk's doesNotUnderstand or what the no-applicable-method does in CLOS.

4.7 Event-Handler vs. Aspect

Although the event-handler paradigm and the aspect paradigm are developed for resolving different issues, their implementation are almost the same, especially from the viewpoint of virtual machine. They both allow programmers to specify

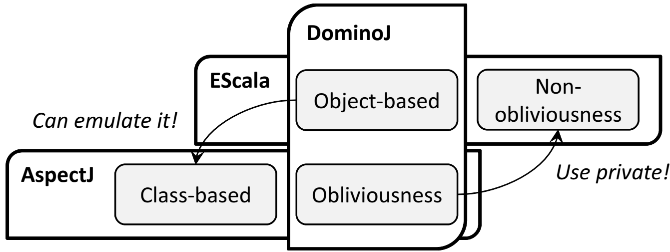


Fig. 9. The design decision of EScala, AspectJ, and DominoJ

which code pieces should be executed after/before the execution of a method. The only difference is the model of specifying and executing the code pieces. First, the behavior of the event-handler paradigm used in EScala is object-based, while the behavior of the aspect paradigm is class-based. Second, in the aspect paradigm used in AspectJ the obliviousness is an important property, but in the event-handler paradigm the non-obliviousness is expected. Obviously, it is impossible to support the contradictory properties at the same time unless we give both constructs into one language. If we just put constructs for the two paradigms into one language, for example providing all syntax of EScala and AspectJ in a new language, it makes the code compilcated and programmers have to learn all of them; it is not what we want to do. Our goal is to make all available by a single construct and let programmers decide how to use it, so we have to choose between object-based behaviors and class-based behaviors, the obliviousness and the non-obliviousness.

The design decisions of EScala, AspectJ, and DominoJ are shown in Figure 9. DominoJ chooses object-based behaviors and the obliviousness since we believe this design is most flexible—the class-based behaviors can be emulated by writing the bindings in the constructors and a method slot can be `private` if the obliviousness is not expected. In this sense, DominoJ can be regarded as either an object-based AOP language or an event mechanism. It does not matter how we call DominoJ since it is just a naming, but here we want to bring up the discussion on the similarities between the event-handler paradigm and the aspect paradigm.

5 Related Work

The delegation introduced by C# [17] allows programmers to declare an event, define its delegate type, and bind a corresponding action to the event. Event composition is also supported by adding a delegate to two or more events. Although the delegate interface hides the executor from the caller, implicit events are not supported. The event must be triggered manually when the change happens. However, C# is able to emulate DominoJ using an unusual programming style: declaring an additional event for every method and always triggering the

event rather than the method. From the point of view, a delegate is very similar to a method slot except the operator += in C# copies the handlers in the event but does not create a reference to the event. However, as in EScala, events and methods are still separate language constructs. Supporting by only one construct means that programmers do not need to decide between using such an unusual style or a normal style at the design stage whether newer modules might regard those methods as events or not. Furthermore, it is annoying that event fields and methods in C# cannot share the same name. Another disadvantage is that we have to ensure that there is at least one delegate for the event before triggering it. Otherwise it will raise an exception. This is not reasonable from the viewpoint of the event mechanism since it just means no one handles the event. In DominoJ no handlers for an event does not raise an exception and the one that triggers an event on a method slot is unaware of handlers.

There are a number of research activities on the integration of object-oriented programming and aspect-oriented programming. Those research use a single dispatch mechanism to unify OOP and AOP and reveal that the integration makes the model clearer, reusable, and compositable. Delegation-based AOP [11,23] elegantly supports the core mechanisms in OOP and AOP by regarding join points as loci of late binding. The model proposed in [12] provides dedicated abstractions to express various object composition techniques such as inheritance, delegation, and aspects. The difference is that DominoJ integrates the event-handler paradigm and the aspect paradigm based on OOP. Another difference is that we propose a new language construct rather than a machine or language model, which makes it compatible with existing object-oriented languages such as Java. Other work such as FRED [21], composition filters [1], predicate dispatching [5], and GluonJ [2] can also be regarded as such integration work.

The method combination in Flavors and CLOS makes related methods easy to combine but not override. By default the combined method in Flavors first calls the before methods in the order that flavors are combined, following by the first primary method, then the after methods in the reverse order. The return value of the combined method is supplied by the primary method, while the return values of the before and after methods are ignored. Similarly, CLOS provides a standard method combination for generic functions. For a generic function call, all applicable methods are sorted before execution in the order the most specific one is first. Besides the primary, before, and after methods, CLOS provides the around methods and `call-next-method` for the primary and around methods. From the viewpoint of method combination, the default closure of a method slot looks like a primary method that can be dynamically added to other method slots as a before or after method, and even as an around method by assigning to the target method slot then using `proceed` as `call-next-method`. It is also easier to express the method combination as a hierarchy in DominoJ.

With regard to the event mechanism, several research activities are devoted to event declaration. Ptolemy [22] is a language with quantified and typed events, which allows a class to register handlers for events, and also allows a handler to be registered for a set of events declaratively. It has the ability to treat the execution

of any expression as an event. The event model in Ptolemy solves the problems in implicit invocation languages and aspect-oriented languages. EventJava [6] extends Java to support event-based distributed programming by introducing the event method, which are a special kind of asynchronous method. Event methods can specify constraints and define the reaction in themselves. They can be invoked by a unicast or broadcast way. Events satisfying the predicate in event method headers are consumed by a reaction. Context-aware applications can be accommodated easily by the mechanism. Both the researchers make events clear and expressive, but they do not support implicit events, which is one of the most significant properties as an event mechanism, whereas DominoJ supports it. Moreover, all events in their model are class-based, so that events for a specified object have to be filtered in the handlers. The binding in DominoJ is object-based, so it can describe the interaction between objects more properly.

On the other hand, several research support the event-handler paradigm upon the aspect paradigm. ECaesarJ [19] introduces events into aspect-oriented languages for context-handling. The events can be triggered explicitly by method calls or defined by pointcuts implicitly. EventCJ [13] is a context-oriented programming language that enables controlling layer activation modularly by introducing events. By declaring events, we can specify when and which instance layer is activated. It also provides layer transition rules to activate or deactivate layers according to events. EventCJ makes it possible to declaratively specify layer transitions in a separate manner. Comparing with DominoJ, using events in the two languages may beak modular reasoning since their event models rely on the pointcut-advice model. Furthermore, events are introduced as a separate construct from methods.

Flapjax [16] proposes a reactive model for Web applications by introducing behaviors and the event streams. Flapjax lets clients use the event-handler paradigm by setting data flows. The handlers for an event can be registered in an implicit way. However, unlike other event mechanisms, it requires programmers to use a slightly different event paradigm. The behavior of DominoJ is more similar to the typical event mechanism while it has the basic ability for the aspect paradigm as well.

Fickle [4] enables re-classification for objects at runtime. Programmers can define several state classes for a root class, create an object at a certain state, and change the membership of the object according to its state dynamically. With re-classification, repeatedly creating new objects between similar classes for an existing object can be avoided. Both Fickle and DominoJ allow to change the class membership of an object at runtime, so other objects holding the identity of the object can be unaware of the changes. The difference is that Fickle focuses on the changes between states while DominoJ focuses on the effect of calling specified methods. Fickle provides better structural ability such as declaring new fields in state classes. However, if the relation between states is not flat and cannot be separated clearly, programmers still have to maintain the same code between state classes. The common code to only part of states can be gathered

up into one class in DominoJ. Furthermore, DominoJ is easier to use for the event-handler paradigm.

The lambda expressions [20] will be introduced in Java 8 as a new feature to support programming in a multicore environment. With the new expression, declaring anonymous classes for containing handlers can be eliminated. The lambda expression of Java 8 is a different construct from methods but method slots can be regarded as a superset of methods.

6 Conclusions

We discussed the similarity between the language constructs for the event-handler paradigm and the aspect paradigm, which motivates us to propose a new language construct, named *method slot*, to support both the paradigms. We presented how a *method slot* is introduced as a language construct in a Java-based language, *DominoJ*. We then discussed how *method slots* can be used for the two paradigms and the coverage of expressive ability. Although the expression of *method slots* is not as rich as other languages, it is much simpler and able to express most functionality in the two paradigms. We also showed its feasibility by implementing a prototype compiler and running a preliminary microbenchmark.

References

1. Bergmans, L., Aksit, M.: Composing crosscutting concerns using composition filters. Commun. ACM 44(10), 51–57 (2001)
2. Chiba, S., Igarashi, A., Zakirov, S.: Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In: OOPSLA 2010, pp. 539–554. ACM (2010)
3. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented programming in AmbientTalk. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 230–254. Springer, Heidelberg (2006)
4. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M., Giannini, P.: *Fickle*: Dynamic object re-classification. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 130–149. Springer, Heidelberg (2001)
5. Ernst, M., Kaplan, C., Chambers, C.: Predicate dispatching: A unified theory of dispatch. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 186–211. Springer, Heidelberg (1998)
6. Eugster, P., Jayaram, K.R.: EventJava: An extension of Java for event correlation. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 570–594. Springer, Heidelberg (2009)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1994)
8. Garlan, D., Jha, S., Notkin, D., Dingel, J.: Reasoning about implicit invocation. In: SIGSOFT 1998/FSE-6, pp. 209–221. ACM (1998)
9. Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: EScala: modular event-driven object interactions in Scala. In: AOSD 2011, pp. 227–240. ACM (2011)

10. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: OOPSLA 2002, pp. 161–173. ACM (2002)
11. Haupt, M., Schippers, H.: A machine model for aspect-oriented programming. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 501–524. Springer, Heidelberg (2007)
12. Havinga, W., Bergmans, L., Aksit, M.: A model for composable composition operators: expressing object and aspect compositions with first-class operators. In: AOSD 2010, pp. 145–156. ACM (2010)
13. Kamina, T., Aotani, T., Masuhara, H.: EventCJ: a context-oriented programming language with declarative event-based context transition. In: AOSD 2011, pp. 253–264. ACM (2011)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
15. Hansen, K.A., Endoh, Y.: A fine-grained join point model for more reusable aspects. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 131–147. Springer, Heidelberg (2006)
16. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for Ajax applications. In: OOPSLA 2009, pp. 1–20. ACM (2009)
17. Microsoft Corporation. C# language specification
18. Microsoft Corporation. Messages and message queues
19. Núñez, A., Noyé, J., Gasiūnas, V.: Declarative definition of contexts with polymorphic events. In: COP 2009, pp. 2:1–2:6. ACM (2009)
20. Oracle Corporation. OpenJDK: Project Lambda, http://openjdk.java.net/projects/lambda/
21. Orleans, D.: Incremental programming with extensible decisions. In: AOSD 2002, pp. 56–64. ACM (2002)
22. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
23. Schippers, H., Janssens, D., Haupt, M., Hirschfeld, R.: Delegation-based semantics for modularizing crosscutting concerns. In: OOPSLA 2008, pp. 525–542. ACM (2008)
24. Smith, R.B., Ungar, D.: Programming as an experience: The inspiration for Self. In: Olthoff, W. (ed.) ECOOP 1995. LNCS, vol. 952, pp. 303–330. Springer, Heidelberg (1995)
25. Sun Microsystems. Abstract Window Toolkit, http://java.sun.com/products/jdk/awt/
26. The AspectJ Project, http://www.eclipse.org/aspectj/
27. The Boost Project. Boost.Signals, http://www.boost.org/libs/signals/
28. The JastAdd Project. JastAddJ: The JastAdd Extensible Java Compiler, http://jastadd.org/web/jastaddj/
29. The Qt Project. Signals & Slots, http://qt-project.org/doc/signalsandslots
30. The SAX project. Simple API for XML, http://www.saxproject.org/
31. The Self project, http://selflanguage.org/
32. The X.Org project. Xlib in X Window System, http://www.x.org/
33. Widom, J., Finkelstein, S.J.: Set-oriented production rules in relational database systems. In: SIGMOD 1990, pp. 259–270. ACM Press (1990)
34. Zhuang, Y., Chiba, S.: Applying DominoJ to GoF Design Patterns. Technical report, Dept. of Math. and Comp., Tokyo Institute of Technology (2011)

Modularity and Dynamic Adaptation of Flexibly Secure Systems: Model-Driven Adaptive Delegation in Access Control Management

Phu H. Nguyen, Gregory Nain, Jacques Klein,
Tejeddine Mouelhi, and Yves Le Traon

Interdisciplinary Centre for Security, Reliability and Trust (SnT)
University of Luxembourg
4 rue Alphonse Weicker, L-2721 Luxembourg
{phuhong.nguyen,gregory.nain,jacques.klein,
tejeddine.mouelhi,yves.letraon}@uni.lu

Abstract. *Model-Driven Security* (MDS) is a specialized *Model-Driven Engineering* (MDE) approach for supporting the development of secure systems. *Model-Driven Security* aims at improving the *productivity* of the development process and *quality* of the resulting secure systems, with *models* as the main artifact. Among the variety of models that have been studied in a *Model-Driven Security* perspective, one can mention *access control* models that specify the access rights. So far, these models mainly focus on static definitions of access control policies, without taking into account the more complex, but essential, *delegation* of rights mechanism. Delegation is a meta-level mechanism for administrating access rights, which allows a user without any specific administrative privileges to delegate his/her access rights to another user. This paper gives a formalization of access control and delegation mechanisms, and analyses the main hard-points for introducing various advanced delegation semantics in *Model-Driven Security*. Then, we propose a modular model-driven framework for 1) specifying access control, delegation and the business logic as separate concerns; 2) dynamically enforcing/weaving access control policies with various delegation features into security-critical systems; and 3) providing a flexibly dynamic adaptation strategy. We demonstrate the feasibility and effectiveness of our proposed solution through the proof-of-concept implementations of different component-based systems running on different adaptive execution platforms, i.e. OSGi and Kevoree.

Keywords: Model-driven security, model-driven engineering, MDE, model composition, delegation, access control, dynamic adaptation, OSGi, Kevoree.

1 Introduction

Software security is a polymorphic concept that encompasses different viewpoints (hacker, security officer, end-user) and raises complex management issues when

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 109–144, 2014.
© Springer-Verlag Berlin Heidelberg 2014

considering the ever-increasing complexity and dynamism of modern software. In this perspective, designing, implementing and testing software for security is a hard task, especially because security is dynamic, meaning that a security policy can be updated at any time and that it must be kept aligned with the software evolution. As one of the key concerns in software security, managing *access control* to critical resources requires the dynamic enforcement of access control policies. Access control policies stipulate actors access rights to internal resources and ensure that users can only access the resources they are allowed to in a given context. A sound methodology supporting such security-critical systems development is extremely necessary because access control mechanisms cannot be "blindly" inserted into a system, but the overall system development must take access control aspects into account. Critical resources could be accessible to wrong (or even malicious) users because of a small error in the specification or in the implementation of the access control policy.

Several design approaches like [24] [4] have been proposed to enable the enforcement of classical security models, such as *Role-Based Access Control* (RBAC) [12] [32]. These approaches bridge the gap from the high-level definition of an access control policy to its enforcement in the running software, automating the dynamic deployment of a given access control policy. Although such a bridge is a prerequisite for the dynamic administration of a given access control policy, it is not sufficient to offer the advanced administration instruments that are necessary to efficiently manage access control. In particular, *delegation* of rights is a complex dimension of access control that has not yet been addressed by the adaptive access control mechanisms. User delegation is necessary for assigning permissions from one user to another user. An expressive design of access control must extensively take into account delegation requirements.

Delegation models based on RBAC management have been characterized as *secure*, *flexible* and *efficient* access management for resource sharing, especially in a distributed environment. Flexible means that different subjects for delegation should be supported, i.e. delegation of roles, specific permissions or obligations. Also, different features of delegation should be supported, like temporary and recurrent delegation, transfer of role or permissions, delegation to multiple users, multi-step delegation, revocation, etc. However, the addition of flexibility for delegation must come with mechanisms to make sure that the security policy of the system is securely consistent. And last but not least, the administration of delegations must remain simple to be efficient. Thus, delegation is a complex problem to solve and to our best knowledge, there has been no complete approach for both specifying and dynamically enforcing access control policies by taking into account various features of delegation. Having such an expressive security model is crucial in order to simplify the administrative task and to manage collaborative work securely, especially with the increase in shared information and distributed systems.

Based on previous work [24], in this paper we propose a new *Modular Model-Driven Security* solution to easily and separately specify 1) the business logic of the system without any security concern using a *Domain Specific Modeling*

Language (DSML) for describing the architecture of a system in terms of components and bindings; 2) the "traditional" access control policy using a DSML based on a RBAC-based metamodel; 3) an advanced delegation policy based on a DSML dedicated to delegation management. In this third DSML, delegation can be seen as a "meta-level" mechanism which impacts the existing access control policies similarly as an aspect can impact a base program. The security enforcement is enabled by leveraging automated model transformation/composition (from security model to architecture model). Consequently, in addition to [24], an advanced model composition is required to correctly handle the new delegation features. In this paper, we claim that delegation needs to be clearly separated from access control because a delegation policy impacts access control rules. Therefore, delegation and access control are not at the same level and should be separated. This separation involves an advanced model composition approach to dynamically know, at any time, what is the set of new access controls that has to be considered, i.e. the "normal" access control rules as well as the access control rules modified by the delegation rules. From a more technical point of view, the security enforcement is dynamically done via automated model transformation/composition (from security model to architecture model) and the dynamic reconfiguration ability of modern adaptive execution platforms.

This paper is an extension of our earlier paper [28] which was presented at the conference Modularity: AOSD'13. The remainder of this paper is organized as follows. Section 2 presents the background on access control, delegation, and the security-driven model-based dynamic adaptation. Formal definitions of our access control model and formalisms of advanced delegation features are given in detail. Section 3 describes a running example. It will be used throughout the paper to show the diverse characteristics of delegation and illustrate the various aspects of our approach. In Section 4, we first give an overview of our approach. Then, we formalize our delegation mechanisms based on RBAC and show how our delegation metamodel can be used to specify expressive access control policies that take into account various features of delegation. Based on the delegation metamodel, we describe our model transformation/composition rules used for transforming and weaving security policy into an architecture model. This section ends with a discussion of several strategies for dynamic adaptation and evolution of security policy. Section 5 describes how our approach has been applied and evaluated in the development of three different systems running on two different adaptive execution platforms. Next, related work is presented in Section 6. Finally, Section 7 concludes the paper and discusses future work.

2 Background

This section introduces the main concepts which are used in this paper. First, formal definitions of *Access Control* and *Delegation* policies are presented. Based on these definitions, some key advanced delegation features are introduced formally. We keep all the definitions here generic so that they can be mapped into different security models like *Role-Based Access Control* (RBAC), *Organization-Based Access Control* (ORBAC) [16], *Discretionary Access Control* (DAC) [19],

etc. These definitions also provide the basis for deriving mutation operators that can be used for testing delegation policy enforcement [29]. Then, a brief summary of previous work on dynamic security policy enforcement [24] is given.

2.1 Access Control

Access Control [14] is known as one of the most important security mechanisms. It enables the regulation of user access to system resources by enforcing access control policies. A policy defines a set of access control rules which expresses: who has the right to access a given resource or not, and the way to access it, i.e. which actions a user can access under which conditions or contexts.

Definition 1 (Access Control). *Let U be a set of users, P be a set of permissions and C be a set of contexts. An access control policy AC is defined as a user-permission-context assignment relation: $AC \subseteq U \times P \times C$. A user u is granted permission p in a given context c if and only if $(u, p, c) \in AC$.*

Additional details about contexts are given in next Section 2.2.

2.2 Delegation

In the field of access control, delegation is a very complex but important aspect that plays a key role in the administration mechanism [5]. A software system which supports delegation should allow its users without any specific administrative privileges to grant some authorizations. Delegation of rights allows a user, called the *delegator*, to delegate his/her access rights to another user, called the *delegatee*. By this delegation, the delegatee is allowed to perform the delegated roles/permissions on behalf of the delegator [9]. The delegator has full responsibility and accountability for the delegated accesses since he/she provides the accesses to the resources to other users, who are not initially authorized by the access control rules to access these resources.

A delegation policy can be considered as an administration-related security policy that is built on top of an access control policy. It is composed of delegation rules that can be specified at two levels: master-level and user-level. Basically, a delegation policy is twofold:

1. It specifies who has the right to delegate which permission (for accessing to a given resource/action/subject) to whom, and in which context. We call this kind of rule *master-level delegation rule* as such a rule is normally defined by security officers. For example, a security officer can define a rule to specify that the head of a department at a university can *only* delegate the permission of updating personnel accounts to a professor.

Definition 2 (Master-level Delegation Policy). *Let U be a set of users, P be a set of permissions and C be a set of contexts. A master-level delegation policy MLD is defined as a user-user-permission-context assignment relation: $MLD \subseteq U \times U \times P \times C$ with the following meaning. A delegation of a permission p from a user u_1 to a user u_2 in a given context c **is allowed** if and only if $(u_1, u_2, p, c) \in MLD$.*

2. It specifies who delegates to whom which permission, and in which context. We call this kind of rule *user-level delegation rule* as these rules are mostly defined by normal users. Note that user-level delegation rules must conform to master-level delegation rules. For example, *Bill* (the head of department) delegates his permission of updating personnel accounts to *Bob* (a professor) during his absence.

Definition 3 (User-level Delegation Policy). *Let U be a set of users, P be a set of permissions and C be a set of contexts. A user-level delegation policy ULD is defined as a user-user-permission-context assignment relation: $ULD \subseteq U \times U \times P \times C$ with the following meaning. A user u_2 **has** a permission p **by delegation** from a user u_1 in a given context c if and only if $(u_1, u_2, p, c) \in ULD$. It can be seen that all the delegations in ULD conform to the rules defined in the MLD. In other words, every delegation at the user-level can only be created if it conforms to the delegation rules defined at the master-level.*

Context. A context is a condition or a combination of conditions in which an access control/delegation rule is active, i.e. enforced in the running system. Cuppens et al. discuss five different kinds of contexts in [10]. These kinds of contexts include temporal context, spatial context, user-declared context, prerequisite context and provisional context. Temporal delegation is delegation within a time constraint, for example delegation is active for two days, or delegation is active for the time the delegator is on vacation. The spatial context relies on the delegator/delegatee's location, e.g. a delegated permission is only active when the delegatee is at office. User-declared context is related to the purpose of the delegator/delegatee, e.g. a delegator may state that his/her delegatee cannot further delegate his/her permissions to someone else. Prerequisite context allows delegation when some precondition is satisfied and the provisional context depends on the previous actions that delegator/delegatee has performed on the system. Moreover, it is possible for a security rule to have a complex context, which is a composition of contexts. Our security model supports context composition using conjunction &, disjunction $\oplus$, and negation $\bar{\ }$.

Note that every access control rule and delegation rule defined in this paper is always associated with a context c. By default, if not specified explicitly, a context c is at least composed of a condition, called *Default*, i.e. always true.

2.3 Advanced Delegation Features

Delegation is a powerful and very useful way to augment access control policy administration. On one hand, it allows users to temporarily modify the access control policy by delegating access rights. By delegation, a delegatee can perform the delegated job, without requiring the intervention of the security officer. On the other hand, the delegator and/or some specific authorized users should be supported to revoke the delegation either manually or automatically. In both cases, the administrative task can be simplified and collaborative work can be

managed securely, especially with the increase in shared information and distributed systems [1]. However, the simpler the administrative task can be, the more complex features of delegation have to be properly specified and enforced in the software system. To the best of our knowledge, there is no approach for both specifying and dynamically enforcing access control policies taking into account all delegation features like temporary delegation, transfer delegation, multiple delegation, multi-step delegation, etc.

In this section, we define the most well-known complex delegation features and formally specify them w.r.t. the definitions of access control and delegation policies. In the following definitions, we use *pre*, *body* and *post* to respectively specify the state of the policy before changing, the state while it is being changed by the function (the delegation rule is being enforced) and the state after changing.

Monotonicity of Delegation. Monotonicity of delegation refers to whether or not the delegator can still use the permission while delegating it [9]. If the delegator can still use the permission while delegating it, the delegation is called grant delegation. Of course, the delegatee can use the permission while it is delegated to him. This is monotonic because available authorizations (in the set AC) are increased due to successful delegation operations. Again, note that every delegation can only be performed if and only if it satisfies the master-level delegation policy.

Definition 4 (Grant Delegation). $grantDelegation(u_1, u_2, p, c) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD$
body $AC := AC \cup \{(u_2, p, c)\}$; $ULD := ULD \cup \{(u_1, u_2, p, c)\}$ *end*
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

Vice versa, if the delegator can not use the permission while delegating it, the delegation is called transfer delegation. As such, this is non-monotonic because available authorizations (in AC) are not increased due to successful delegation operations.

Definition 5 (Transfer Delegation). $transferDelegation(u_1, u_2, p, c) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD$
body $AC := AC \setminus \{(u_1, p, c)\}$; $AC := AC \cup \{(u_2, p, c)\}$; $ULD := ULD \cup \{(u_1, u_2, p, c)\}$ *end*
post $(u_1, p, c) \notin AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

Temporary Delegation. This is also a very common feature of delegation needed by users. When revocation is handled automatically, the delegation is called temporary. In this case, the delegator specifies the temporal conditions in which this delegation applies: only at a given time, after or before a given time, or during a given time interval. The temporal conditions may correspond to a day of the week, or to a time of the day, etc. If the temporal context is not used, the delegation needs to be revoked manually.

Definition 6 (Temporary Delegation). *Let c be a given context of a delegation (either grant delegation or transfer delegation). A delegation is specified as temporary if its context c is associated with a time constraint. The delegation will only be active while the time constraint is satisfied.*

For example, if the context is *vacation period*, a delegator *Bill* could have an associated delegation rule with the following context:
$c := c\&vacation_period(startDate, endDate)$
where $vacation_period(startDate, endDate) : -$
$startDate \leq endDate \wedge afterDate(startDate) \wedge beforeDate(endDate)$
Here, $afterDate(date)$ returns *true* iff *date* is equal or later than the current date. Similarly, $beforeDate(date)$ returns *true* iff *date* is equal or earlier than the current date.

Multiple Delegation. A permission can be delegated to more than a delegatee at a given time. However, the number of times that a permission is concurrently delegated have to be controlled. Multiple delegation refers to the maximum number of times that a permission can be delegated at a given time.

Definition 7 (Multiple Delegation). *Let N_m be the maximum number of times that a permission can be concurrently delegated. N_m is predefined by the security officer. The number of concurrent delegations in which the same role or permission is delegated at a given time, in a given context cannot exceed N_m.*

We introduce a counting function to count the number of delegations of a permission which is delegated by a delegator in a given context. The number returned by this function is always updated according to the change in the delegation policy, i.e. the number of delegation rules related to permission p.
$countDelegation(u, p, c) := |\{(u, v, p, c) \mid \forall v \in U : (u, v, p, c) \in ULD\}|$
If the number of concurrent delegations of the same permission at a given time, in a given context has not exceeded N_m, then this permission is still allowed to be delegated.
$grantDelegation(u_1, u_2, p,$
$c\&countDelegation(u_1, p, c) < N_m) : -$ pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD \wedge countDelegation(u_1, p, c) < N_m$
body $AC := AC \cup \{(u_2, p, c)\}$; $ULD := ULD \cup \{(u_1, u_2, p, c)\}$ end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

Multi-step Delegation. This characteristic refers to the maximum number of steps (N_s, normally specified by a security officer) that a permission p can be re-delegated, counted from the first delegator of this permission. So if $N_s = 0$ that means the permission p can not be re-delegated anymore.

Definition 8 (Multi-step Delegation). *Let $N_s \geq 0$ be the maximum number of steps that a permission p can be re-delegated. A permission p can only be delegated iff $N_s > 0$.*

First, let us define a helper function that returns the number of times a permission p is re-delegated in a given context c. $stepCounter(u_0, p, c) := N_s$ where u_0 is the first delegator of p in the delegation chain: u_0 delegates p to ... in a given context c; ... re-delegates p to u_1 in context c; and u_1 re-delegates p to u_2 in context c. Here, "..." is the users in the middle of the delegation chain, u_1 is the current last delegatee of this chain, and u_2 is the next delegatee if $stepCounter(u_1, p, c) \geq 1$.

If there exists a predefined maximum number of steps N_s for a permission p as described above, the delegation is specified as following.

$grantDelegation(u_1, u_2, p, c$
$\& stepCounter(u_1, p, c) \geq 1) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \in MLD \wedge stepCounter(u_1, p, c) \geq 1$
body $AC := AC \cup \{(u_2, p, c)\}; ULD := ULD \cup \{(u_1, u_2, p, c)\}; stepCounter(u_2, p, c)$
$:= stepCounter(u_1, p, c) - 1$ end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$

Delegation Revocation. Delegation supports a revocation feature in which a delegation can be revoked and permissions are returned to the original user.

Definition 9 (Delegation Revocation). *Delegation revocation is the ability for any delegation can be manually revoked by authorized users.*

The revocation of a grant delegation means to deny access of the delegatee to the delegated permission.

Definition 10. $revokeGrantDelegation(u_1, u_2, p, c) : -$
pre $(u_1, p, c) \in AC \wedge (u_2, p, c) \in AC \wedge (u_1, u_2, p, c) \in ULD$
body $AC := AC \setminus \{(u_2, p, c)\}$; $ULD := ULD \setminus \{(u_1, u_2, p, c)\}$ *end*
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \notin ULD$

The permission to be revoked is deleted from the access rights of the delegatee. To revoke a transfer delegation, it is not only to deny access of the delegatee to the delegated permission but also to re-grant access to the delegator who is temporarily not having this access.

Definition 11. $revokeTransferDelegation(u_1, u_2, p, c) : -$
pre $(u_2, p, c) \in AC \wedge (u_1, p, c) \notin AC \wedge (u_1, u_2, p, c) \in ULD$
body $AC := AC \setminus \{(u_2, p, c)\}$; $AC := AC \cup \{(u_1, p, c)\}$; $ULD := ULD \setminus \{(u_1, u_2, p, c)\}$
end
post $(u_1, p, c) \in AC \wedge (u_2, p, c) \notin AC \wedge (u_1, u_2, p, c) \notin ULD$

We have presented formal definitions of access control, delegation and various delegation features. These definitions are generic (at the conceptual level) so that they can be mapped into different security models like RBAC, ORBAC, DAC, etc. Section 4 shows how these formal concepts can be implemented (based on RBAC) using MDE techniques.

2.4 Security-Driven Model-Based Dynamic Adaptation

In [24], the authors have proposed to leverage MDE techniques to provide a very flexible approach for managing access control. The different steps of this approach are summed up in Figure 1. On one hand, access control policies are defined by security experts, using a DSML, which describes the concepts of access control, as well as their relationships. On the other hand, the application is designed using another DSML for describing the architecture of a system in terms of components and bindings. This component-based software architecture only contains the business components of the application, which encapsulate the functionalities of the system, without any security concern. Then, the authors define mappings between both DSMLs describing how security concepts are mapped to architectural concepts. These mappings are used to fully generate an architecture that enforces the security rules. When the security policy is updated, the architecture is also updated. Finally, the proposed technique leverages the notion of *models@runtime* [21] in order to keep the architectural model (itself synchronized with the access control model) synchronized with the running system. This way, the running system can be dynamically updated in order to reflect changes in the security policy. Only users who have the right to access a resource can actually access this resource.

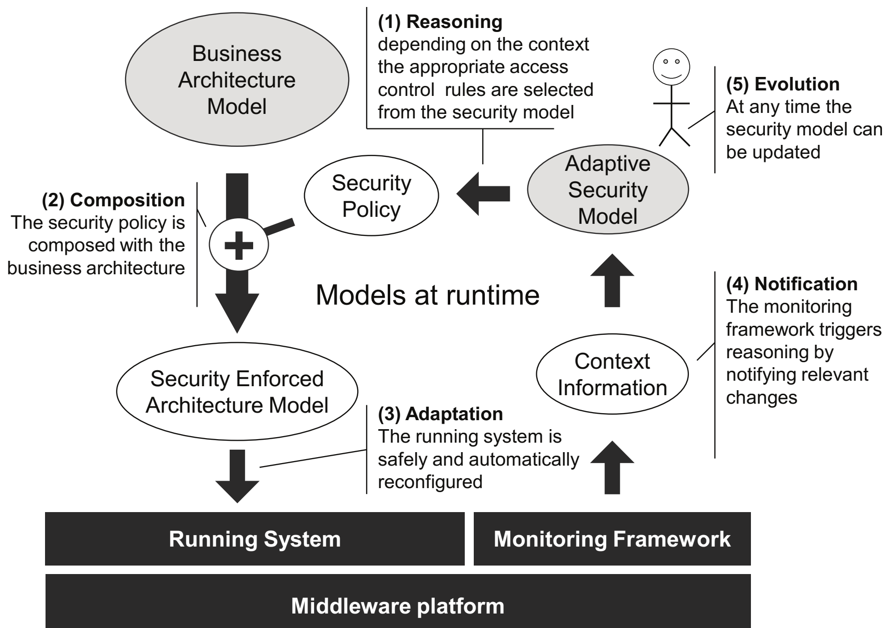


Fig. 1. Overview of the Model-Driven Security Approach of [24]

3 A Running Example

In this section, we give a motivating example which will be used throughout the paper for describing the diverse characteristics of delegation and illustrating the various aspects of our approach.

Let us consider a *Library Management System* (LMS) providing library services with security concerns like access control and delegation management. There are two types of user accounts: *personnel accounts* (director, secretary, administrator and librarian) are managed by an administrator; and *borrower accounts* (lecturer and student) are managed by a *secretary.* The *director* of the library has the same accesses as a secretary, but additionally, he can also consult the personnel accounts. The *librarian* can consult the borrower accounts. A secretary can add new books in the LMS when they are delivered. *Lecturers* and *students* can borrow, reserve and return books, etc. In general, the library is organized with the following entities and security rules.

Roles (users): access rights (e.g. working days)
Director (*Bill*): consult personnel account, consult, create, update and delete borrower account.
Secretary (*Bob* and *Alice*): consult, create, update and delete borrower account, add book.
Administrator (*Sam* and *Tom*): consult, create, update and delete personnel account.
Librarian (*Jane* and *John*): consult borrower account, find book by state, find book by keyword, report a book damaged, report a book repaired, fix a book.
Lecturer (*Paul*) and **Student** (*Mary*): find book by keyword, reserve, borrow and return book.

Resources and actions to be protected
Personnel Account: consult, create, update and delete personnel account.
Borrower Account: consult, create, update and delete borrower account.
Book: report a book damaged, report a book repaired, borrow a book, deliver a book, find book by keyword, find book by state, fix a book, reserve a book, return a book.

In this organization, users may need to delegate some of their authorities to other users. For instance, the director may need the help of a secretary to replace him during his absence. A librarian may delegate his/her authorities to an administrator during a maintenance day.

It is possible to only specify role or action delegations by using the DSML described in [24]. For instance, a role delegation rule can be created to specify that *Bill*, the director (prior to his vacation) delegates his role to *Bob*, one of his secretaries. But it is impossible for *Bill* to define whether or not *Bob* can re-delegate the *director* role to someone else (in case *Bob* is also absent for some reason). The role delegation of *Bill* to *Bob* is also handled manually: it is enforced when *Bill* creates the delegation rule and only revoked when *Bill* deletes this

rule. There is no way for *Bill* to define a temporary delegation where its active duration is automatically handled. Obviously the DSML described in [24] is not expressive enough to specify complex characteristics of delegation.

There are many delegation situations that should be supported by the system. We give some delegation situations of the LMS as follows:
1. The director (*Bill*) delegates his role to a secretary (*Bob*) during his vacation (the delegation is automatically activated at the start of his vacation and revoked at the end of his vacation).
2. A secretary (*Alice*) delegates her task/action of create borrower account to a librarian (*Jane*).
3. A secretary (*Bob*) transfers his role to an administrator (*Sam*) during maintenance day. In case of a transfer delegation, the delegator temporarily loses his/her rights during the time of delegation.
4. The role administrator is not delegable.
5. The permission of deleting borrower account is not delegable.
6. The director can delegate, on behalf of a secretary, the secretary's role (or some his/her permitted actions) to a librarian (e.g. during the secretary's absence).
7. If a librarian empowered in role *secretary* by delegation is no longer able to perform this task, then he/she can delegate, again, this role to another librarian.
8. The secretary empowered in role *director* by delegation is not allowed to delegate/transfer, again, this role to another secretary.
9. A secretary is allowed to delegate his/her role to a librarian only and to one librarian at a given time.
10. A secretary is allowed to delegate his/her task of book delivery to a librarian only and scheduled on every Monday.
11. *Bill* can delegate his role and permitted actions only to *Bob*
12. *Bob* is not allowed to delegate his role.
13. *Alice* is not allowed to delegate her permitted action of book delivery.
14. Users can always revoke their own delegations.
15. The director can revoke users from their delegated roles.
16. A secretary can revoke librarians empowered in *secretary* role by delegation, even if he/she is not the creator of this delegation (e.g. the creator is the director or another librarian).

This running example shows the two levels of delegation rules as defined in the previous section: user-level (rules defined by a user: e.g. situations 1, 2, 3) and master-level (rules defined by a security officer: e.g. 4, 5, 6). Obviously, delegation rules at user-level have to conform to rules at master-level. For example, the security officer can define that users of role *director* are able to delegate on behalf of users of role *secretary*. Then at user-level, *Bill* (director) can create a delegation rule to delegate, on behalf of *Alice*, her role (secretary) to *Jane* (librarian).

4 Model-Driven Adaptive Delegation

4.1 Overview of Our Approach

In our approach, as noted in Section 2, delegation is considered as a "meta-level" mechanism which impacts the existing access control policies, like an aspect can impact a base program. We claim that to handle advanced delegation rules, an ideal solution is to logically separate the delegation rules from the access control policy, each being specified in isolation, and then compose/weave them together to obtain a new access control policy (called active security policy) reflecting the delegation-driven policy (Figure 2). We present our metamodel (DSML) for specifying delegation based on RBAC in Section 4.2.

The separation of concerns is not only between delegation and access control, but also between the security policy and the business logic of the system. Figure 3 presents a wider view of the overall approach. In order to enforce a security policy for the system, the core business architecture model of the system is composed with the active security policy previously obtained. The architecture model is expressed in another DSML, called architecture metamodel (an architecture modeling language described in [24]). The idea is to reflect security policy into the system at the architecture level. Section 4.3 defines transformation rules to show how security concepts are mapped into architectural concepts.

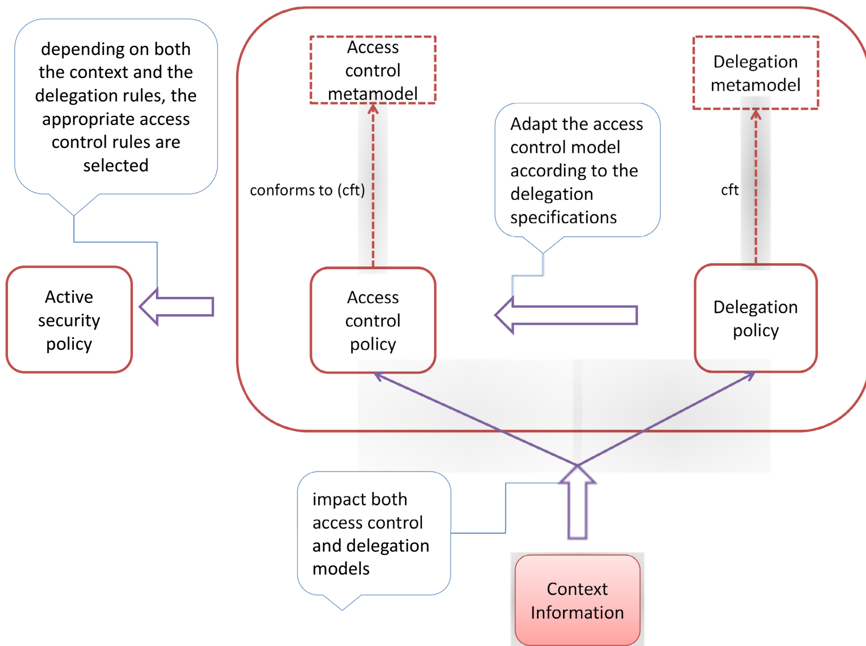


Fig. 2. Delegation impacting Access Control

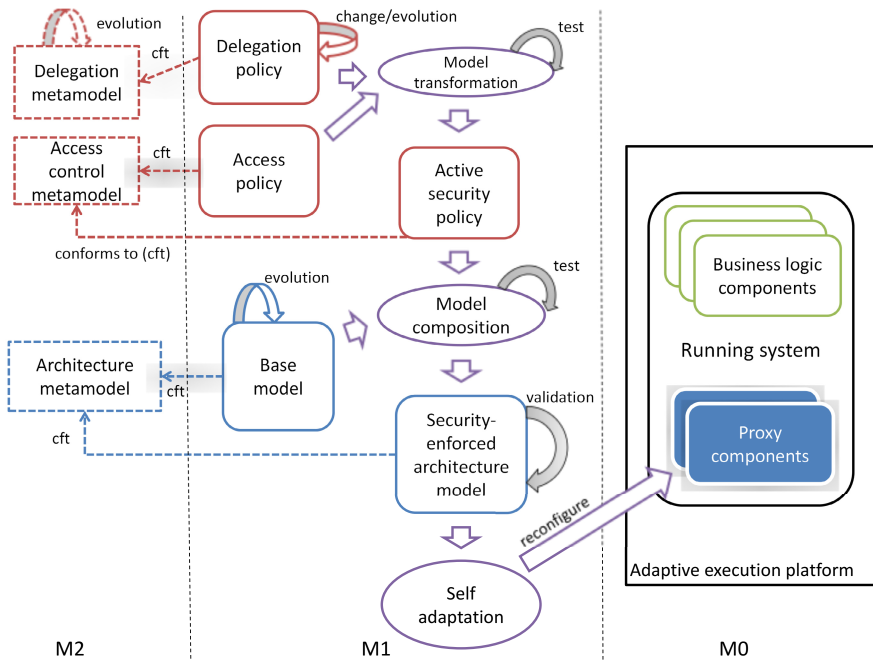


Fig. 3. Overview of our approach

The security-enforced architecture model obtained above is a pure architecture model which by itself reflects how the security policy is enforced in the system. Our model-driven framework to reflect security policy at the architecture model is *generic*, meaning that from the security-enforced architecture model of the system, it is possible to enforce security policy for running systems on different execution platforms. In Section 5, we show how our approach is applied for two different adaptive execution platforms, i.e. OSGi [33] and Kevoree [13][1]. It is important to note that the security-enforced architecture model is not used for generating the whole system but only the proxy components. These proxy components can be adapted and integrated with the running system at runtime to physically enforce the security policy. The adaptation and integration can be done by leveraging the runtime adaptation mechanisms provided by modern adaptive execution middleware platforms. The approach of possibly generating proxy components overcomes some main limitations of [24]. Section 4.4 is dedicated to discuss our strategy for adaptation and evolution of the secure systems.

4.2 Delegation Metamodel

Our metamodel, displayed in Figure 4, defines the conceptual elements and their relationships that can be used to specify access control and delegation policies

[1] `www.kevoree.org`, last access October 2013.

which are defined in Section 2. Because the delegation mechanism is based on RBAC, we first explain the main conceptual elements of role-based access control. Then, we show how our conceptual elements of delegation, based on the RBAC conceptual elements, can be used to specify various delegation features which are defined in Section 2.

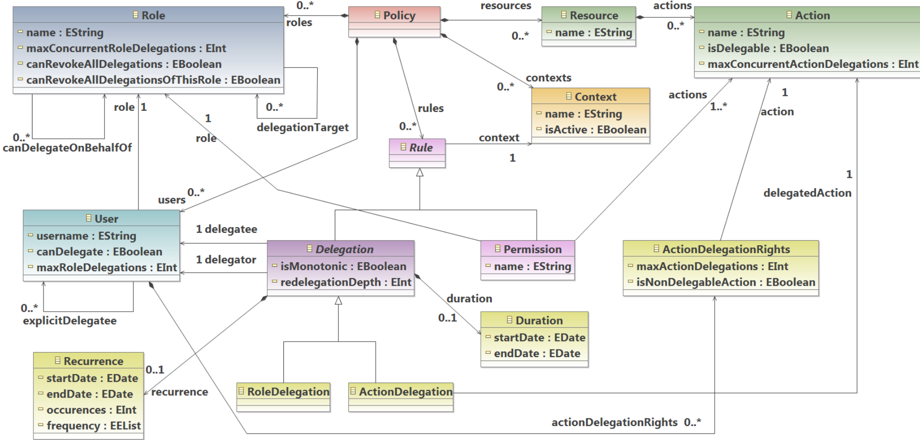


Fig. 4. The Delegation metamodel

As shown in Figure 4, the root element of our metamodel is the *Policy*. It contains *Users*, *Roles*, *Resources*, *Rules* and *Contexts*. Each user has one role. A security officer can specify all the roles in the system, e.g. *admin*, *director*, etc., via the *Role* element. In order to specify an access control policy, the security officer should have defined in advance the *resources* that must be protected from unauthorized access. Each resource contains some *actions* which are only accessible to authorized users. These protections are defined in rules: *permission rules* and *delegation rules*. Permission rules are used to specify which actions are accessible to users based on their *roles*. That means, without delegation rules or user-specific access control rules, every user is able to access the actions associated with his/her role only. Delegation rules are used to specify which actions are accessible to users by delegation. There are two basic types of delegation:

– **Role delegation:** When users empowered in role(s) delegated by other user(s), they are allowed to access not only actions associated with their roles but also actions associated with the delegated role(s).

– **Action delegation:** Instead of delegating their roles, users may want to delegate only some specific actions associated with their roles.

Another important aspect of our access control and delegation framework is the notion of *context* which has been introduced in Section 2.2. It can be seen

from our metamodel that every permission/delegation rule is associated with a context. A rule is only active within its context. The concept of context actually provides our model with high flexibility. Security policies can be easily adapted according to different contexts.

The full metamodel for specifying delegation is displayed in Figure 4. It depicts the features that are supported by our delegation framework. All delegation management features are developed based on two basic types of delegation mentioned above. In the following, we show how the delegation features can be specified, w.r.t. our metamodel. In other words, this is how the formal definitions in Section 2 are actually implemented.

- **Temporary delegation:** This is one of the most common types of delegation used by users. It describes when the delegation *starts* to be active and when it *ends*. The delegator can specify that the delegated role/action is authorized only during a given time interval, e.g. situation 1 of the running example in Section 3. Actually, this can be specified using the recurrence of delegation described below, but we want to define it separately because of its common use.
- **Monotonicity (Transfer of role or permissions):** A property *isMonotonic* can be used to specify if a delegation is monotonic or non-monotonic. The former (*isMonotonic* = *true*) specifies that the delegated access right is available to both the delegator and delegatee after enforcing this delegation. As defined in Section 2, this delegation is called a grant delegation. The latter (*isMonotonic* = *false*) means the delegated role/action is transferred to the delegatee, and the delegator temporarily loses his rights while delegating, e.g. situation 3. In this case, the delegation is called a transfer delegation.
- **Recurrence:** It refers to the repetition of the delegation. A user may want to delegate his role to someone else for instance every week on Monday. Recurrence defines how the delegation is repeated over time. It is similar to what is implemented in calendar system and more precisely the icalendar standard (RFC2445[2]). It has several properties; the *startDate* and *endDate* are the starting and ending dates of the recurrence. In addition, the *startDate* defines the first occurrence of the delegation. The *frequency* indicates one of the three predefined types of frequency, daily, weekly or monthly. The *occurrences* is the number of times to repeat the delegation. If the *occurrences* is for instance equals to 2 it means that it should only be repeated twice even when the *endDate* is not reached. An example of this delegation is situation 10 of the running example.
- **Delegable roles/actions:** These kinds of delegation define which roles or actions can be delegated and how (master-level). A policy officer can specify that a role can only be delegated/transferred to specific role(s), e.g. situation 9. If no *delegationTarget* is defined for a role, this role cannot be delegated/-transferred, e.g. situation 4. If a role or action (*isDelegable* = *false*) is not delegable, it should never be included in a delegation rule. Moreover, a role can also be delegated by a user not having this role but his/her own role is

[2] http://www.rfc-editor.org/info/rfc2445

specified as can delegate on behalf of a user in this role (*canDelegateOnBehalfOf = true*), e.g. situation 6.

- **Multiple delegations:** It should be possible to define the max number of concurrent delegations in which the same role or action can be delegated at a given time (master-level delegation rule). The properties *maxConcurrentRoleDelegations* and *maxConcurrentActionDelegations* define how many concurrent delegations of the same role/action can be granted, e.g. situation 9. Moreover, it is possible to define for each specific user a specific maximum number of concurrent delegations of the same role/action: *maxRoleDelegations* and *maxActionDelegations*.
- **User-specific delegation rights:** All user-specific elements are used to define more strict rules for a specific user rather for his/her role. There are other user-specific delegations than *maxRoleDelegations* and *maxActionDelegations*. It is possible to define that a specific user is allowed to delegate his role/permitted action(s) or not (*canDelegate = true* or *false*), e.g. situation 12. The property *isNonDelegableAction* specifies an action that a specific user cannot delegate, e.g. situation 13. Moreover, the security officer can define to which explicit user(s) only (*explicitDelegatee*) a user can delegate/-transfer his role to, e.g. situation 11.
- **Multi-step delegation:** It provides flexibility in authority management, e.g. situations 7, 8. The property *redelegationDepth* is used to define whether or not the role/action of a delegation can be delegated again. When a creator creates a new delegation, he/she can specify how many times the delegated role/action can be re-delegated. If the *redelegationDepth* = 0, it means that the role/action cannot be delegated anymore, e.g. situation 8. If the *redelegationDepth* > 0, it means the role/action can be delegated again and each time it is re-delegated, the *redelegationDepth* is decreased by 1.
- **Revocations:** All users can revoke their own delegations, e.g. situation 14. Security officer may set *canRevokeAllDelegations = true* for a role with a super revocation power in such a way that a user empowered in this role can revoke all delegations, e.g. situation 15. Moreover, a role can also be defined such that every user empowered in this role can revoke any delegation from this role (*canRevokeAllDelegationsOfThisRole = true*), even he/she is not the delegator of the delegation, e.g. situation 16.

Moreover, each possible instance of the security policy has to satisfy all necessary validation condition expressed as OCL invariants. For example, we can make sure that no delegation is out of target, meaning that delegatee's role has to be a delegation target of delegator's role:

context Delegation **inv** NoDelegationOutOfTarget: self.delegator.role.delegationTarget −>exists (t | t = self.delegatee.role)

Or to check that for every user, the number of concurrent role delegations cannot be over its thresholds:

context User **inv** NoRoleDelegationOverMax: RoleDelegation.allInstances −>select (d | d.delegator = self) −>size() ≤ self.role.maxConcurrentRoleDelegations and RoleDelegation.allInstances −>select (d | d.delegator = self) −>size() ≤ self.maxRoleDelegations

Other examples are to restrict the value of the *redelegationDepth* must not be negative, or *startDate* cannot be later than *endDate*:

context Delegation **inv** NonNegativeDeleDepth: self.redelegationDepth ≥ 0

context Duration **inv** ValidDates: self.startDate $\leq$ self.endDate

4.3 Transformations/Compositions

After specifying a security policy by the DSML described in Section 4.2, it is crucial to dynamically enforce this policy into the running system. Transformations play an important role in the dynamic enforcement process. Via model transformations, security models containing delegation rules and access control rules are automatically transformed into component-based architecture models. Note that instances of security models and architecture models are checked before and after model transformations, using predefined OCL constraints.

The model transformation is executed according to a set of transformation rules. The purpose of defining transformation rules is to correctly reflect security policy at the architectural level. Based on transformation rules, security policy is automatically transformed to proxy components, which are then integrated to the business logic components of the system in order to enforce the security rules. The metamodel of component-based architecture can be found in [24] and an instance of it can be seen in Figure 7. We first describe the transformation that derives an access control model according to delegation rules (step 1), and then describe another transformation to show how security policy can be reflected at the architecture level (step 2). Moreover, we also show an alternative way of transformation that combines two steps into one.

Adapting Role-Based Access Control policy model to reflect delegation (step 1): Within the security model shown in Figure 2, delegation rules are considered as "meta-level" mechanisms that impact the access control rules. The appropriate access control rules and delegation rules are selected depending on the context information and/or the request of changing security rules coming from the system at runtime. According to the currently active context (e.g. WorkingDays), only *in-context* delegation rules and *in-context* access control rules of the security model (e.g. rules that are defined with context = WorkingDays) are taken into account to derive the active security policy model (Figure 2). Theoretically, we could say that delegation rules impact the core RBAC elements in the security model in order to derive a pure RBAC model (without any delegation and context elements) which conforms to a "pure" metamodel of RBAC (Figure 5). Delegation elements of a security policy model are transformed as follows:

A.1: Each **action delegation** is transformed into a new permission rule. The *subject* of the permission is *user* (delegatee) object. The set of *actions* of the permission contains the delegated action.

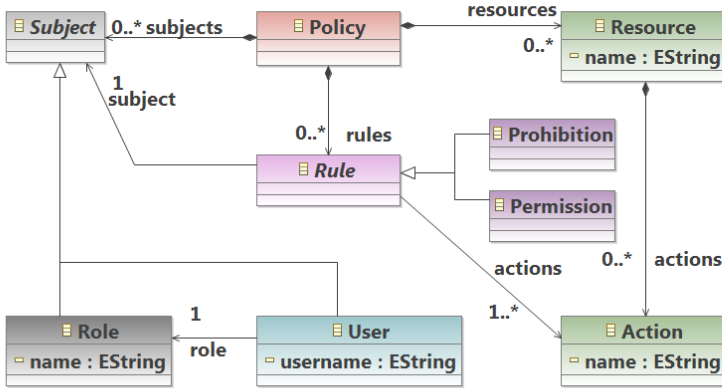


Fig. 5. A pure RBAC metamodel

A.2: Each **role delegation** is transformed as follows. First, a set of actions associated to a role is identified from the permissions of this role. Then, each action is transformed into a permission like transforming an **action delegation** described above.
A.3: A **temporary delegation** is only taken into account in the transformation if it is in active duration defined by the **start** and **end** properties. In fact, when its active duration starts the (temporary) action/role delegation is transformed into permission rule(s) as described above. When its active duration ends the temporary delegation is removed from the policy model.
A.4: If an action delegation is of type transfer delegation (**monotonic**), then it is transformed into a permission rule and a prohibition rule. The *subject* of the permission is the *user*-delegatee object. The set of *actions* of the permission contains the delegated action. The *subject* of the prohibition is the *user*-delegator object. The set of *actions* of the prohibition contains the delegated action.
A.5: If a role delegation is of type transfer delegation, then it is also transformed into a permission rule and a prohibition rule. The *subject* of the permission is the *user*-delegatee object. The set of *actions* of the permission contains the delegated actions. The delegated actions here are the actions associated with this role. The *subject* of the prohibition is the *user*-delegator object. The set of *actions* of the prohibition also contains the delegated actions.
A.6: If a delegation rule is defined with a **recurrence**, based on the values set to the recurrence, the delegation rule is only taken into account in the transformation within its *fromDate* and *untilDate*, repeated by *frequency* and limited by *occurrences*. In other words, only active (during recurrence) delegation rules are transformed.
A.7: (User-specific) If a user is associated with any **non-delegable action**, the action delegation containing this action and this user (as delegator) is not transformed into a permission rule. Similarly, if a user is specified as he/she **cannot delegate** his/her role/action, no role/action delegation involving this user is transformed.

A.8: (Role/action-specific) Any delegation rule with a **non-delegable** role/action will not be transformed. In fact, a delegation rule is only transformed if it satisfies (at least) both user-specific and role/action-specific requirements.
A.9: Only a role delegation to a user (delegatee) whose role is in the set of **delegationTarget** will be considered in the transformation.
A.10: Before any delegation is taken into account in the transformation, it has to satisfy the requirements of **max concurrent action/role delegations**. Note that the user-specific values have higher priorities than the role-specific values.
A.11: A delegation is only transformed if its **redelegationDepth** > 0. Whenever a user empowered in a role/an action by delegation re-delegates this role/action, the newly created delegation is assigned a **redelegationDepth** = the previous **redelegationDepth** - 1.

After transforming all delegation rules, we obtain a pure RBAC model which reflects both the delegation model and access control model. This pure RBAC model is then transformed into a security-enforced architecture model as described next.

Transformation of Security Policy to Component-Based Architecture (step 2): The transformation rules are defined below. The goal is to transform every security policy model (pure RBAC model obtained in step 1) which conforms to the metamodel shown in Figure 5 to a component-based architecture model which conforms to the metamodel described in [24]. However, both the security policy model and the *base model* provided by a system designer are used as inputs for the model transformation/composition. Via a graphical editor, the security designer must define in advance how the resource elements in the policy model are related to the business components in the *base model*. Figure 6 shows how each action in the policy can be mapped to the Java method in the business logic.

Because the *base model* already conforms to the architecture metamodel, we now only focus on transforming the security policy model into the security-reflected architecture model. As we know, this transformation/composition process will also weave the security-reflected elements into the base model in order to obtain the security-enforced architecture model.

The core elements of RBAC like *resource*, *role*, and *user* are transformed following these transformation rules. All the transformation rules make sure that the security policy is reflected at the architectural level.

R-A.1: Each *resource* is transformed into a component *instance*, called a *resource* proxy component. According to the relationship between the resource elements in the policy model and the business components in the *base model*, each *resource* proxy component is connected to a set of business components via bindings. To be more specific, each action of a resource element is linked to an operation of a business component (Figure 6). By connecting to business components, a *resource* proxy component provides and requires all the services (actions) offered by the resource.
R-A.2: Each *role* is also transformed into a *role* proxy component. According to the granted accesses (permission rules associated with this role) to the services

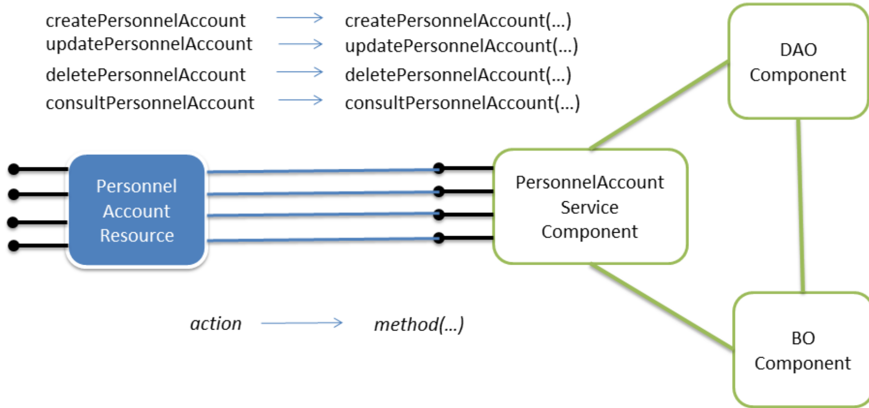


Fig. 6. Mapping Resources to Business Logic Components

provided by the resources, the corresponding *role* proxy component is connected to some *resource* proxy component(s) (Figure 7). A *role* proxy component is connected to a *resource* proxy component by transforming granted accesses into ports and bindings. Each (active) access granted to a role is transformed into a pair of ports: a client port associated with the *role* proxy component, a server port associated with the *resource* proxy component, and a binding linking these ports.

R-A.3: Each *user* element defined in the policy model is also transformed into a *user* proxy component. Because each user must have one role, each *user* proxy component is connected to the corresponding *role* proxy component. However, each user may have access to actions associated to not only his/her role but also to actions associated to other roles by delegation. Thus, each *user* proxy component may connect to several *role* proxy components. The connection is established by transforming each access granted to a user into a pair of ports: a client port associated with the *user* proxy component, a server port associated with the corresponding *role* proxy component (providing the access/port), and a binding linking these ports (Figure 7). Actually, the granted accesses are calculated not only from *permission* rules but also from *prohibition* rules. Simply, the granted accesses that equal permissions exclude prohibitions.

In our approach, revocation of a delegation simply consists in deleting the corresponding delegation rule. In this way, the revocation is reflected at the architectural level and physically enforced in the running system. Moreover, both the delegator and delegatee elements will be removed if these users are not involved in any delegation rules. As described above, user elements are transformed into proxy components. However, it is important to stress that only users involved in delegation rules (e.g. Bill, Bob and Sam in Figure 7) are created in the security policy model and transformed into proxy components. Users who are not involved in any delegation rules (e.g. Jane and Mary in Figure 7), are manipulated as session objects which directly access the services offered by the corresponding role proxy components.

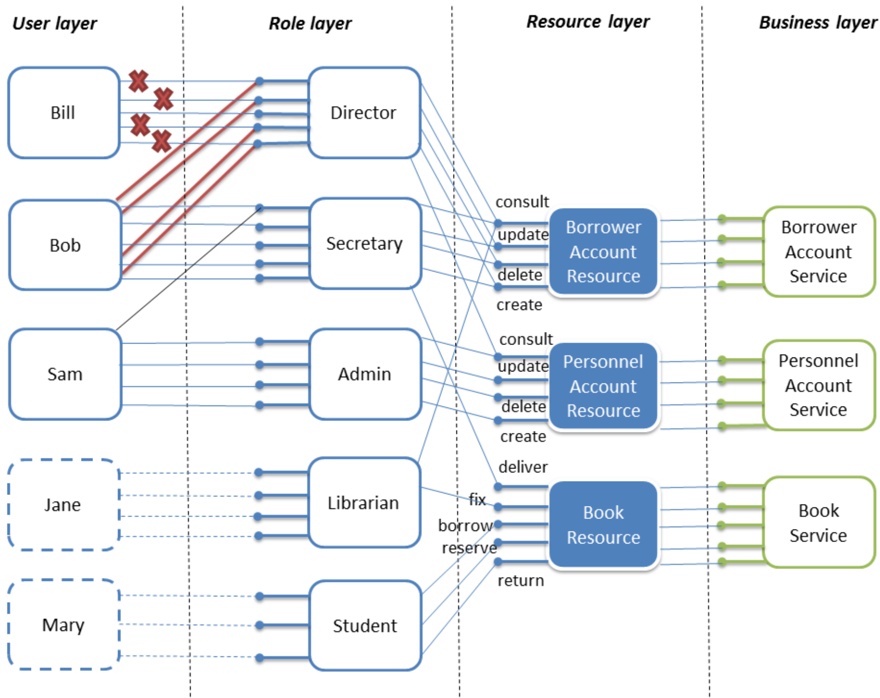


Fig. 7. Architecture reflecting security policy before and after adding a delegation rule (bold lines)

Two steps described above are two separate model transformations that are mainly used to explain how delegation can be considered as a "meta-level" mechanism for administrating access rights. The first model transformation is to transform a delegation-driven security model into a pure RBAC model. The second model transformation is to transform the RBAC model into an architecture model. In fact, these two steps could be done in only one model transformation that directly transforms the delegations, the access control policy and the business logic model into an architecture model reflecting the security policy. However, this alternative way (described in the following) has the disadvantage of losing the intermediate security model (the active security policy) that could be useful for traceability purpose.

An Alternative Way Using only One Transformation: In this approach, we have to define different transformation rules to transform directly every security policy model which conforms to the metamodel, shown in Figure 4, to a component-based architecture model which conforms to the architecture metamodel described in [24]. Core elements of RBAC like *resources*, *roles* and *users* are transformed following these transformation rules:
R-B.1: Each *resource* is transformed into a component *instance*, called a *resource* proxy component (already presented).

R-B.2: Each *role* also is transformed into a *role* proxy component (already presented). The only difference here is that the *context* has to be taken into account (in the step 2 of transformation mentioned earlier, no *context* existed because *context* was already dealt with in the step 1). Because every permission is associated with a *context*, we only transform permissions with the *context* that is active at the moment.
R-B.3: Each *user* element defined in the policy model is also transformed into a *user* proxy component. However, the connection (via bindings) from a *user* proxy component to the *role* proxy component(s) is not only dependent on the user's role but also delegation rules that the corresponding user involved in. The transformation of delegation rules is presented below.

All the transformation rules above make sure that access control rules are reflected at the architecture level. However, the delegation rules will impact this transformation process in order to derive the security-enforced architecture model reflecting both access control and delegation policy. Delegation elements of a policy model are transformed as follows:
R-B.4: Each action involved in an **action delegation** is transformed into a pair of ports and a binding. A client port (representing the required action) is associated with the user (*delegatee*) proxy component. The binding links the client port to the corresponding server port (representing the same action provided) that associated with the *role* proxy component reflecting the role of the *delegator*.
R-B.5: Each **role delegation** is transformed in a similar way as **action delegation**. First, a set of actions associated to a role can be identified from the permissions of this role. Then, each action in the set is transformed into a pair of ports and a binding as transforming an action delegation.
R-B.6: A **temporary delegation** is only transformed into bindings if it is still in active duration defined by **start** and **end** properties.
R-B.7: If a delegation is of type **transfer delegation**, then both user elements (delegator and delegatee) are transformed into delegator and delegatee proxy components as described above. The delegator proxy component is not connected to the corresponding role proxy component because he/she already transfered his/her access rights to the delegatee. Figure 7 shows a change in the architecture when *Bill* transfers his role to *Bob*.
R-B.8: If a delegation is defined with a **recurrence**, based on the values set to recurrence, the delegation rule is only active during the recurrence (similar to A.6).
R-B.9: If a user is associated with any **non-delegable action**, the delegation of this action is not taken into account while doing the transformation. Similarly, if a user is specified as he/she **can not delegate** his/her role/action, no delegation requested by this user will be transformed.
R-B.10: Only a role delegation to a user (delegatee) whose role is in the set of **delegationTarget** will be consider in the transformation.
R-B.11: Before any delegation is taken into account in the transformation, it has to satisfy the requirements of **max concurrent action/role delegations**.

Note that the user-specific values have higher priorities than the role-specific values.
R-B.12: A delegation is only transformed if its **redelegationDepth** > 0. Whenever a user empowered in a role/an action by delegation re-delegates this role/action, the newly created delegation is assigned a **redelegationDepth** = the previous **redelegationDepth** - 1.

By taking into account delegation rules while transforming access control rules of policy model into security-enforced architecture model, both delegation and access control rules are reflected at the architecture level.

4.4 Adaptation and Evolution Strategies

The model transformation/composition presented in Section 4.3 ensures that the security policies are correctly and automatically reflected in an architectural model of the system. The key steps to support delegation (i.e. specifications and transformations) are already presented in Sections 4.2 and 4.3. The last step consists in a physical enforcement of the security policy by means of a dynamic adaptation of the running system. In this section, our adaptation and evolution strategies are discussed.

Adaptation. The input for the adaptation process is a newly created security-enforced architecture model (Figure 8). First, this new architecture model is validated using invariant checking [22]. This valid architectural model actually represents the new system state the runtime must reach to enforce the new security policy of the system. According to the classical MAPE control loop of self-adaptive applications, our reasoning process performs a comparison (using EMFCommpare) between the new architecture model (target configuration) and the current architecture model (kept synchronized with the running system) [23]. This process triggers a code generation/compilation process, and also generates a safe sequence of reconfiguration commands [22]. Actually, the code generation/-compilation process is only triggered if there are new proxy components, e.g. new user proxy components involved in delegation, that need to be introduced into the running system. The dynamic adaptation of the running system is possible thanks to modern adaptive execution platforms like OSGi [33] or Fractal [8], and most recently Kevoree [13], which provide low-level APIs to reconfigure a system at runtime. The running system is then reconfigured by executing the safe sequence of commands, compliant to the platform API, issued by the reasoning process. In an optimized model@runtime platform like Kevoree, all we need to do is to provide the reconfiguration script (Kevoree script) for the platform. The reasoning process is taken care of by the platform. In fact, the generation/compilation phase if needed could be time-consuming. However, this phase has no impact on the running system, which remains stable until being adapted by executing the reconfiguration script. Thus, the actual adaptation phase lasts for only several milliseconds.

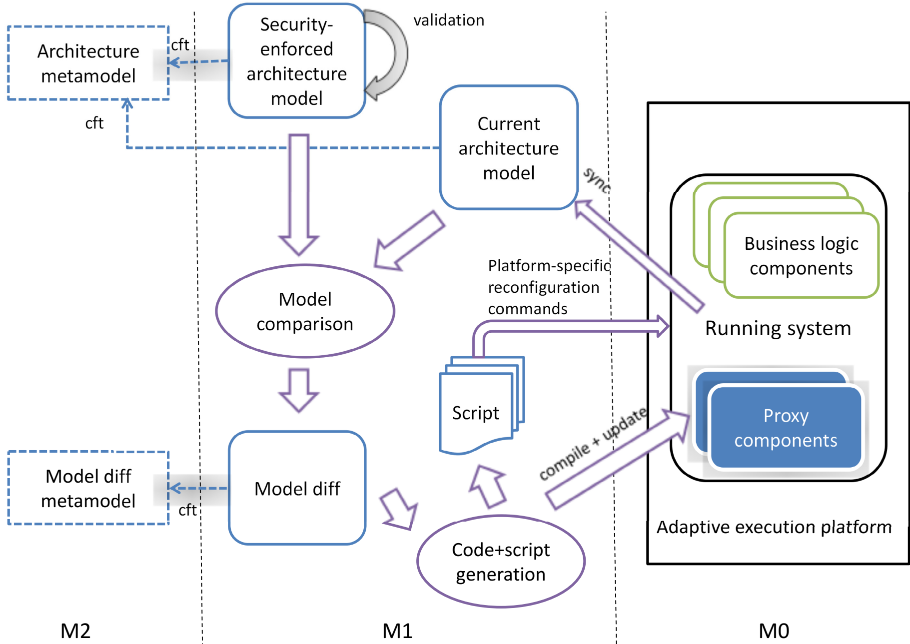


Fig. 8. Overview of our adaptation strategy

In [24], the adaptation is entirely based on executing platform-specific reconfiguration scripts specifying which components have to be stopped and which components and/or bindings should be added and/or removed. This results in several limitations regarding delegation mechanisms:

L.1: Using only reconfiguration scripts implies to create all the potentially needed ports (used for bindings between *user* proxy components) beforehand. But all the combinations of users, roles, resources, actions could lead to a combinatorial explosion and make it infeasible for implementation.

L.2: In [24], the delegation between users are reflected using bindings connecting one *user* proxy component to another. But this approach is not suitable for supporting complex delegation features. For example, a transfer delegation will be reflected by adding bindings between the *delegator* and *delegatee* but removing bindings between *delegator* and the corresponding *role* proxy component. Consequently both *delegator* and *delegatee* cannot access the resource, which does not correctly reflect a transfer delegation.

L.1 can be solved by the automatic re-generation of proxy components and bindings between them according to changes in the architectural model. Moreover, as mentioned in Section 4.3, only users involved in a delegation are transformed into *user* proxy components with necessary ports and bindings. In this way, only required ports and bindings are created dynamically. **L.2** is solved by our model transformation approach. All complex delegation features are

considered as "meta-level" mechanisms that impact access control rules. In this way, a transfer delegation will be reflected by adding bindings between the *delegatee* and the corresponding delegated *role* proxy component, but removing bindings between *delegator* and the corresponding *role* proxy component.

Our adaptation strategy could take more time than simply running a reconfiguration script because of the generation and compilation time of newly generated proxy components. But the process of generating and compiling new proxy components does not in fact harm the performance because each proxy component is very light-weight and only necessary proxy components are generated (see Section 5). Moreover, for each specific security policy, it is possible to think in advance and prepare as many proxy component types as possible. This strategy could make the generation/compilation phase unnecessary for most of the cases, except some major evolution of the business logic and/or the security policy.

Evolution. In [24], the evolution of the security policy is not totally dealt with. It is possible to run a reconfiguration script to reflect changes like adding, removing and updating rules. But adding a new user, role or resource requires the generation and compilation of new proxy components, which is impossible using only reconfiguration scripts. Thus, our strategy of automatically generating and compiling proxy components (see Section 5) is more practical w.r.t. evolution.

Another important aspect of evolution relates to the addition, removal or update of resources and actions in the business logic. The base architecture model can be updated with changes in the business logic, e.g. when a new resource is added. On the other side, security officers can manually update the mappings (Figure 6) following changes of resources/actions in the base architecture model. By composing the security model with the base architecture model as described earlier, the security policy is evolved together with the business logic of the system.

5 Implementation and Evaluation

This section shows how the steps described in Figure 3 have been implemented. In order to prove that our approach is generic, we target two different adaptive execution platforms: OSGi (Section 5.1) and Kevoree (Section 5.2). Figure 9 shows that our metamodels and model-to-model transformation/composition are generic, i.e. independent of execution platforms. Only the adaptation process (e.g. the reconfiguration script) and the running system are platform-specific. We evaluate our proof-of-concept implementations and discuss the results in Section 5.3. The description of three case studies used in our experiments are given below. The business logic of these case studies are the same for the OSGi and Kevoree adaptive execution platforms.

To evaluate the feasibility of our approach, we have applied it on three different Java-based case studies, which have also been used in our previous research work on access control testing [25]:

1) LMS: as described in our running example.

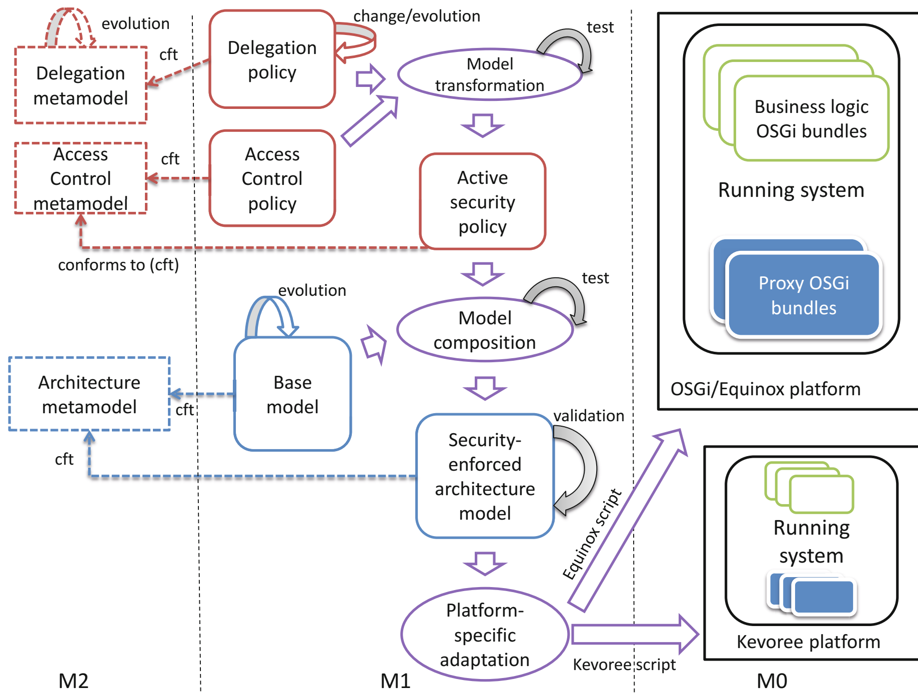


Fig. 9. OSGi and Kevoree as adaptive execution platforms

2) VMS[3]**:** The Virtual Meeting System offers simplified web conference services. The virtual meeting server allows the organization of work meetings on a distributed platform. When connected to the server, a user can enter (exit) a meeting, ask to speak, eventually speak or plan new meetings. There are three resources (Meeting, Personnel Account, User Account) and six roles (Administrator, Webmaster, Owner, Moderator, Attendee and Non-attendee) defined for this system with many access control rules, and delegation situations between the users of each role.
3) ASMS: The Auction Sale Management System allows users to buy and sell products online. Each user in the system has a profile including some personal information. Users wanting to sell a product (sellers) are able to start a new auction by submitting a description of the product, the starting and ending date of the auction. There are five resources (Sale, Bid, Comment, Personnel Account, User Account) and five roles (Administrator, Moderator, Seller, Senior Buyer, and Junior Buyer) defined for this system, also with many access control rules, and delegation situations between users of each role.

Table 1 provides some information about the size of these three systems (the number of classes, methods and lines of code). In terms of security policies,

[3] For more information about VMS (server side), please refer to `http://franck.fleurey.free.fr/VirtualMeeting`.

Table 1. Size of each system in terms of source code

	# Classes	# Methods	# LOC
LMS	62	335	3204
VMS	134	581	6077
ASMS	122	797	10703

Table 2. Security rules defined for each system

	# AC rules	# Delegations	Total
LMS	23	4	27
VMS	36	8	44
ASMS	89	8	97

Table 2 shows the number of access control (AC) rules and delegation rules defined for each system, used in our experiments.

All these systems are designed as component-based systems. The business components of each system contain the business logic, e.g. Book Service component, Personnel Account component, Meeting, Sale, Authenticate component, Data Access Object components, etc. To enable dynamic security enforcement for a system, the resources (components that have to be controlled) are specified in the base model, and mapped to the resources of the security policies. Our metamodels are applicable for different systems without any modification or adaptation. The structure of delegation and access control policies for all case studies is the same, only roles, users, resources, actions are specific to each case study. The proxy components are automatically generated and synchronized with the security policy model via model transformations and reconfiguration at runtime. The model-to-model transformation and model-to-text transformation (code generation) can be implemented correspondingly using transformation engines like Kermeta [26] (or ATL[4]), and Xpand [18].

5.1 OSGi (Equinox) as the Target Adaptive Execution Platform

As shown in Figure 9, once we obtain the security-enforced architecture model from the previous steps, we have to reflect this security enforcement in the running system. In case we use Equinox[5] as the target execution platform, all components (business logic components and proxy components) are implemented as OSGi bundles (Spring Dynamic Modules) [30]. In OSGi service platforms, there are two ways to declare and bind services via interfaces (ports): declaring/binding exported services in Spring osgi-context.xml files, or in the source code by overriding the method *start* of *BundleActivator* class of OSGi bundle. Here we show the code for the sake of simplicity but in practice, the declaration of services

[4] `http://www.eclipse.org/atl/`
[5] `http://www.eclipse.org/equinox/`

and bindings can be configured in XML files which means no need to recompile code to change the bindings. Once the services are made available, they can be called from other services. For example, the code snippet in Listing 1.1 shows how the *deleteBorrowerAccountService* of a proxy component of Role *Director* is bound to the exported service reference of the *deleteBorrowerAccountService* of the *BorrowerAccountResource* proxy component (lines 1-8). The lines 12-20 show that this Role *Director* can also access to *consultPersonnelAccount* of *PersonnelAccountResource*.

```
1  ServiceReference[] refIdeleteBorrowerAccount_DIRECTOR =
       bundleContext.getServiceReferences(
               lms.proxy.interfaces.IdeleteBorrowerAccount.class
3                  .getName(), "(host=BorrowerAccountResource)");
   lms.proxy.interfaces.IdeleteBorrowerAccount
       serverIdeleteBorrowerAccount_DIRECTOR = (lms.proxy.interfaces
       .IdeleteBorrowerAccount) bundleContext
5          .getService(refIdeleteBorrowerAccount_DIRECTOR[0]);

7  myDIRECTORService
           .setdeleteBorrowerAccountService(
       serverIdeleteBorrowerAccount_DIRECTOR);
9
   ...
11
   ServiceReference[] refIconsultPersonnelAccount_DIRECTOR =
       bundleContext.getServiceReferences(
13             lms.proxy.interfaces.IconsultPersonnelAccount.class
                   .getName(), "(host=PersonnelAccountResource)");
15 lms.proxy.interfaces.IconsultPersonnelAccount
       serverIconsultPersonnelAccount_DIRECTOR = (lms.proxy.
       interfaces.IconsultPersonnelAccount) bundleContext
           .getService(refIconsultPersonnelAccount_DIRECTOR[0]);
17
   myDIRECTORService
19         .setconsultPersonnelAccountService(
       serverIconsultPersonnelAccount_DIRECTOR);
```

Listing 1.1. Services and Bindings in the Director proxy component

As mentioned before, all the proxy components are very lightweight components. Every method of proxy components only contains the redirecting call to another service that (directly/indirectly) calls to the real method in the business logic. The code snippet in Listing 1.2 shows that a call to the *deleteBorrowerAccount* method (line 1) of a proxy component of Role *Director* actually is redirected to call the *deleteBorrowerAccount* method (line 3) of the *BorrowerAccountResource* proxy component that already was made available previously (lines 1-8, Listing 1.1). Similarly, the *consultPersonnelAccount* method (line 7) contains a call to the *consultPersonnelAccount* method (line 10) of the *PersonnelAccountResource* proxy component that already was made available previously (lines 12-20, Listing 1.1).

```
1  public void deleteBorrowerAccount(
        lms.bo.user.BorrowerAccount borrowerAccount) throws
      BSException {
3     deleteBorrowerAccountService.deleteBorrowerAccount(
      borrowerAccount);
   }
5  ...

7  public lms.bo.user.PersonnelAccount consultPersonnelAccount(
        lms.bo.user.User personnel) throws BSException {
9     return consultPersonnelAccountService
           .consultPersonnelAccount(personnel);
11 }
```

Listing 1.2. Redirecting the method calls in the Director proxy component

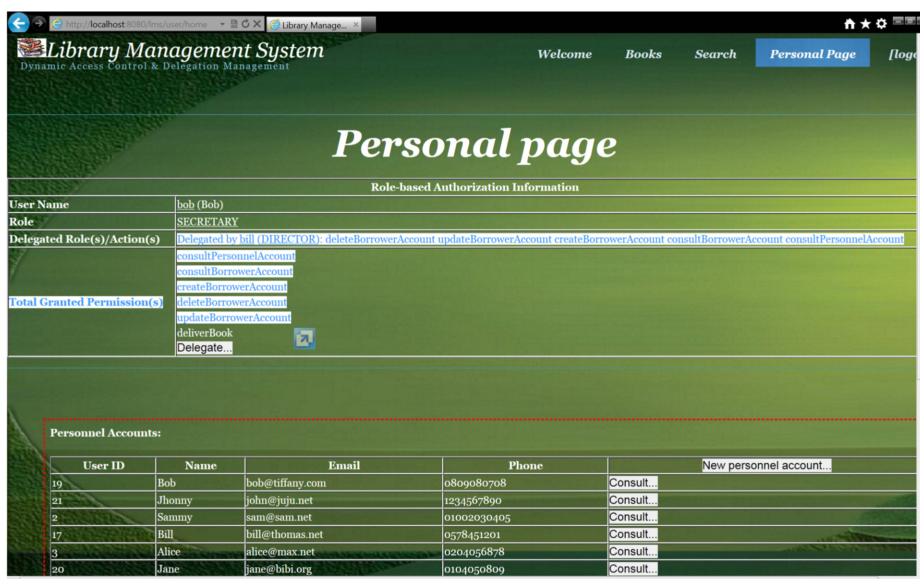


Fig. 10. *Bob* is delegated by *Bill* 5 permissions, e.g. *consult personnel account*

The adaptation process is directed by a generated reconfiguration script that is specific for Equinox adaptive execution platform. The reconfiguration script is executed in order to reflect the change of the policy from the model level to the running system, e.g. a new delegation rule is active. Figure 10 shows a new delegation rule has been enforced in the running system so that *Bob* (*Secretary*) is delegated the permission to *consult personnel account* by *Bill* (*Director*). This means after enforcing this delegation rule, there exists a connection from the port *consultPersonnelAccount* of the User proxy component *Bob*, via corresponding Role and Resource proxy components, to the real method *consultPersonnelAccount* in the business logic.

5.2 Kevoree as the Target Adaptive Execution Platform

In case we use Kevoree as the target execution platform, all components (business logic components and proxy components) are implemented as Kevoree component instances [13]. The adaptation process is driven by a generated reconfiguration Kevoree script. The Kevoree script orchestrates the adaptation process of the running system by adding, removing component instances, binding the service ports between proxy components. An example of the configuration of proxy components (the 3-layer architecture) is shown in Figure 7. In order to explain how the proxy components are implemented, let us take a look at the *Director* Role proxy component. This Role proxy component is representative as it is between the User layer and the Resource layer (Figure 7). It can be seen that this *Director* Role proxy component provides the ports (services) to the User proxy components and also requires the ports (services) of the Resource proxy components. The code snippet in Listing 1.3 shows the ports required and provided by the *Director* Role proxy component. The required ports are bound to the corresponding ports provided by the *BorrowerAccountResource* proxy component and the *PersonnelAccountResource* proxy component. The provided ports are to be bound by the ports required by the corresponding User proxy components.

Once the corresponding User proxy component calls the service provided by the port “deleteBorrowerAccountIn” (line 8, Listing 1.3), the method *deleteBorrowerAccount* (line 3, Listing 1.4) in the *Director* Role proxy component is executed that in turn calls the service provided by the port “deleteBorrowerAccountOut” (line 5, Listing 1.4. In fact this port is provided by the *BorrowerAccountResource* proxy component that finally calls to the corresponding method of *deleteBorrowerAccount* in the business logic code.

```
1  @Requires({
           @RequiredPort(name="deleteBorrowerAccountOut", type =
       PortType.SERVICE, className = IDeleteBorrowerAccount.class,
       optional = true),
3          @RequiredPort(name = "consultPersonnelAccountOut", type =
        PortType.SERVICE, className = IconsultPersonnelAccount.class
       , optional = true),
           ...
5  })

7  @Provides({
           @ProvidedPort(name="deleteBorrowerAccountIn", type =
       PortType.SERVICE, className =
9  IDeleteBorrowerAccount.class),
           @ProvidedPort(name = "consultPersonnelAccountIn", type =
       PortType.SERVICE, className = IconsultPersonnelAccount.class)
       ,
11         ...
   })
```

Listing 1.3. The ports required and provided by the *Director* Role proxy component

```
@Override
@Port(name = "deleteBorrowerAccountIn", method = "
    deleteBorrowerAccount")
public void deleteBorrowerAccount(BorrowerAccount borrowerAccount
    ) throws BSException {

    IDeleteBorrowerAccount deleteBorrowerAccountPort =
    getPortByName("deleteBorrowerAccountOut",
    IDeleteBorrowerAccount.class);

    deleteBorrowerAccountPort.deleteBorrowerAccount(
    borrowerAccount);
}
```

Listing 1.4. Redirecting the method call in the *Director* Role proxy component in Kevoree

There are three main advantages of using Kevoree over OSGi as the execution platform. First, all we need to provide for the platform is the Kevoree reconfiguration script saying how to adapt the system. The Kevoree execution platform takes care of the necessary adaptation order for the running system according to changes. In case of using OSGi, we have to take care of the adaptation order manually. Secondly, the *model@runtime* environment of Kevoree makes it easier for implementing our model driven framework. In Kevoree, we can use the Kevoree framework itself to manage the security policy models. Thirdly, the way of declaring ports and bindings in Kevoree are very close to the concepts of ports and bindings described in our 3-layer architecture (Figure 7). This makes it very convenient to implement the running systems in Kevoree.

Table 3. Performance of weaving Security Policies using Kermeta and ATL

	# Rules	Kermeta 1.4.1	Kermeta 2.0.6	ATL 3.2.1
LMS	27	4s	1.836s	0.048s
VMS	44	7s	2.161s	0.055s
ASMS	97	18s	2.834s	0.140s

5.3 Evaluation and Discussion

There are two kinds of response time we would like to measure in our case studies: the authorization mechanism and the dynamic adaptation according to changing security policies. The experiments were performed on Intel Core i7 CPU 2.20 GHz with 2.91 GB usable RAM running on Windows 7. The number of security rules defined for each system in our experiments is indicated in Table 2. Because all our access control and delegation rules are transformed into proxy components reflecting our security policy, response times to an access request only depends on method calls between these proxy components and business components (Figure 7). Unsurprisingly, response time to every resource access

is a constant, only about 1 millisecond, because the access is already possible or not by construction. In other words, our 3-layered architecture reflecting security policy enables very quick response, independently from the number of access control and delegation rules.

For experimenting with performance of adapting the running system, we have implemented the model transformation/composition rules using not only Kermeta but also ATL. Regarding the adaptation process, Table 3 shows results of each case study for performing the model transformations of security policies mentioned in Table 2, using Kermeta 1.4.1, Kermeta 2.0.6 and ATL 3.2.1 correspondingly. Note that these model-to-model transformations are generic, platform-independent w.r.t the implementation platform of the running system. Thus, the same model-to-model transformations are used in both cases of implementation platform, i.e. OSGi and Kevoree. At first, we used Kermeta 1.4.1 to implement our model transformations. However, the performance of using Kermeta 1.4.1 shown in Table 3 was disappointing. It took more than 18 seconds to weave 97 security rules in case of the ASMS. To know if this performance problem is inherently linked to our approach or simply linked to the use of Kermeta 1.4.1, we decided to also implement our model transformations using ATL 3.2.1. Our experiments show that the implementation using ATL 3.2.1 is much more efficient. We can conclude that the initial performance issue was due to Kermeta 1.4.1. Then, we have tried to use Kermeta 2.0.6 which is the latest version of Kermeta at this moment, compiled to byte code, which means much better performances. As can be seen from Table 3, the results of using Kermeta 2.0.6 are better compared to using Kermeta 1.4.1.

Note that the transformation, code generation and compilation are performed "offline" meaning that the running system is not yet adapted. The actual adaptation happens when the newly compiled proxy components are integrated into the running system to replace the current proxy components. This actual adaptation process takes only some milliseconds by using the low-level APIs to reconfigure a system at runtime provided by the modern adaptive execution platforms, i.e. OSGi [33] and Kevoree [13]. Right after the new proxy components are up and running, the new security policy is really enforced in the running system.

6 Related Work

There is substantial work related to delegation as an extension of existing access control models. Most researchers focused on proposing models solely relying on the RBAC formalism [32], which is not expressive enough to deal with all delegation requirements. Therefore, some other researchers extended the RBAC model by adding new components, such as new types of roles, permissions and relationships [2,34,1,9,27]. In [5], the authors proposed yet another delegation approach for role-based access control (more precisely for ORBAC model) which is more flexible and comprehensive. However, no related work has provided a model-driven approach for both specifying and dynamically enforcing access control policies with various delegation requirements. Compared to [24], we extend the

model-based dynamic adaptation approach of [24] with some key improvements. More specifically, we propose a new DSML for delegation management, but also new composition rules to weave delegation in an RBAC-based access control policy. In addition, we present a new way (by generating proxy) to implement the adaptation of the security-enforced architecture of the system. Indeed, we provide an extensive support for delegation as well as co-evolution of security policy and security-critical system. That means our approach makes it possible to deeply modify the security policy (e.g. according to evolution of the security-critical system) and dynamically adapt the running system, which is often infeasible using the other approaches mentioned above.

In addition, several researchers proposed new flexible access control models that may not include delegation, but allow a flexible and easy to update policy. For instance, Bertino *et al.* [6] proposed a new access control model that allows expressing flexible policies that can be easily modified and updated by users to be adapted to specific contexts. The advantage of their model resides in the ability to change the access control rules by granting or revoking the access based on specific exceptions. Their model provides a wide range of interesting features that increase the flexibility of the access control policy. It allows advanced administrative functions for regulating the specification of access controls rules. More importantly, their model supports delegation, enabling users to temporarily grant other users some of their permissions. Furthermore, Bertolissi *et al.* proposed DEBAC [7] a new access control model based on the notion of event that allows the policy to be adapted to distributed and changing environments. Their model is represented as a term rewriting system [3], which allows specifying changing and dynamic access control policies. This enables having a dynamic policy that is easy to change and update.

As far as we know, no previous work tackled the issue of enforcing adaptive delegation. Some previous approaches were proposed to help modelling more general access control formalisms using UML diagrams (focusing on models like RBAC or MAC). RBAC was modelled using a dedicated UML diagram template [17], while Doan *et al.* proposed a methodology [11] to incorporate MAC in UML diagrams during the design process. All these approaches allow access control formalisms to be expressed during the design. They do not provide a specific framework to enable adaptive delegation at runtime. Concerning the approaches related to applying MDE for security, we can cite UMLSEC [15], which is an extension of UML that allows security properties to be expressed in UML diagrams. In addition, Lodderstedt *et al.* [20] propose SECUREUML which provides a methodology for generating security components from specific models. The approach proposes a security modelling language to define the access control model. The resulting security model is combined with the UML business model in order to automatically produce the access control infrastructure. More precisely, they use the Meta-Object facility to create a new modelling language to define RBAC policies (extended to include constraints on rules). They apply their technique in different examples of distributed system architectures including Enterprise Java Beans and Microsoft Enterprise Services for .NET. Their approach provides a

tool for specifying the access control rules along with the model-driven development process and then automatically exporting these rules to generate the access control infrastructure. However, they do not directly support delegation. Delegation rules should be taken into account early and the whole system should be generated again to enforce the new rules. Our approach enables supporting directly the delegation rules and dynamically enforcing them by reconfiguring the system at runtime.

7 Conclusion and Future Work

In this paper, we have proposed an extensive *Model-Driven Security* approach for adaptive delegation in access control management. By giving a formalization of access control and delegation mechanisms, we introduced various advanced delegation features that would provide secure, flexible, and efficient access control management. It has been shown that these advanced delegation features can be specified using our delegation DSML. Our DSML supports complex delegation characteristics like temporary, recurrence delegation, transfer delegation, multiple and multi-step delegation, etc. We have also shown that revocation can be dealt with in a simple manner. Another main contribution of this paper is our adaptive delegation enforcement in which delegation is considered as a "meta-level" mechanism that impacts the access control rules. A complete model-driven framework has been proposed to enable dynamic enforcement of delegation and access control policies that allows the automatic configuration of the system according to the changes in delegation/access control rules. Moreover, our framework also enables an adaptation strategy that better supports co-evolution of security policy and business logic of the system. The model-driven framework proposed in this paper can be applied for securing (distributed) systems running on different adaptive execution platform like OSGi (Equinox), or an optimized models@runtime framework such as Kevoree. Our approach has been validated via three different case studies with consideration of performance and extensibility issues.

In this paper, we only focus on the delegation of rights, further work will also be dedicated to the delegation of obligations and the support for usage control [31]. Usage control is called the next generation of access control with more flexible access management mechanisms that we would adopt our current approach for. We have not dealt with this idea yet in this paper, but keep it for our future work. Moreover, revocation mechanism in our current approach has not been completely taken into account, i.e. without options of strong/weak revocation. Besides, in order to complete the framework, we also propose an approach for testing delegation policy enforcement. In this direction, we continue working on the extension of testing delegation policy enforcement via mutation analysis [29].

Acknowledgments. We would like to thank the anonymous referees for their comments and suggestions. This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, under the MITER project C10/IS/783852.

References

1. Ahn, G.-J., Mohan, B., Hong, S.-P.: Towards secure information sharing using role-based delegation. J. Netw. Comput. Appl. 30(1), 42–59 (2007)
2. Barka, E., Sandhu, R.: Role-based delegation model/hierarchical roles (RBDM1). In: Proceedings of the 20th Annual Computer Security Applications Conference, ACSAC 2004, pp. 396–404. IEEE Computer Society (2004)
3. Barker, S., Fernández, M.: Term rewriting for access control. In: DBSec, pp. 179–193 (2006)
4. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: From UML models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. 15(1), 39–91 (2006)
5. Ben-Ghorbel-Talbi, M., Cuppens, F., Cuppens-Boulahia, N., Bouhoula, A.: A delegation model for extended RBAC. Int. J. Inf. Secur. 9(3), 209–236 (2010)
6. Bertino, E., Jajodia, S., Samarati, P.: A flexible authorization mechanism for relational data management systems. ACM Trans. Inf. Syst. 17(2), 101–140 (1999)
7. Bertolissi, C., Fernández, M., Barker, S.: Dynamic event-based access control as term rewriting. In: DBSec, pp. 195–210 (2007)
8. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.: The Fractal component model and its support in Java. Software Practice and Experience, Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems 36(11-12), 1257–1284 (2006)
9. Crampton, J., Khambhammettu, H.: Delegation in role-based access control. International Journal of Information Security 7(2), 123–136 (2008)
10. Cuppens, F., Cuppens-Boulahia, N.: Modeling contextual security policies. International Journal of Information Security 7(4), 285–305 (2007)
11. Doan, T., Demurjian, S., Ting, T.C., Ketterl, A.: MAC and UML for secure software design. In: FMSE 2004: Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering, pp. 75–85. ACM (2004)
12. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. 4(3), 224–274 (2001)
13. Fouquet, F., Nain, G., Morin, B., Daubert, E., Barais, O., Plouzeau, N., Jézéquel, J.-M.: An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 87–101. Springer, Heidelberg (2012)
14. Jajodia, S., Samarati, P., Sapino, M.L., Subrahmanian, V.S.: Flexible support for multiple access control policies. ACM Trans. Database Syst. 26(2), 214–260 (2001)
15. Jürjens, J.: UMLsec: Extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)
16. Kalam, A.A.E., Baida, R.E., Balbiani, P., Benferhat, S., Cuppens, F., Deswarte, Y., Miege, A., Saurel, C., Trouessin, G.: Organization based access control. In: Proceedings of IEEE 4th International Workshop on Policies for Distributed Systems and Networks, POLICY 2003, pp. 120–131 (2003)
17. Kim, D.-K., Ray, I., France, R.B., Li, N.: Modeling role-based access control using parameterized UML models. In: Wermelinger, M., Margaria-Steffen, T. (eds.) FASE 2004. LNCS, vol. 2984, pp. 180–193. Springer, Heidelberg (2004)
18. Klatt, B.: Xpand: A closer look at the model2text transformation language. Language (10/16/2008) (2007)

19. Lampson, B.W.: Protection. SIGOPS Oper. Syst. Rev. 8(1), 18–24 (1974)
20. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
21. Morin, B., Barais, O., Jézéquel, J.-M., Fleurey, F., Solberg, A.: Models@ Run.time to support dynamic adaptation. Computer 42(10), 44–51 (2009)
22. Morin, J.-M.B., Barais, O., Nain, G., Jézéquel: Taming dynamically adaptive systems with Models and Aspects. In: ICSE 2009: 31st International Conference on Software Engineering (May 2009)
23. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.-M., Solberg, A., Dehlen, V., Blair, G.S.: An aspect-oriented and model-driven approach for managing dynamic variability. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 782–796. Springer, Heidelberg (2008)
24. Morin, B., Mouelhi, T., Fleurey, F., Le Traon, Y., Barais, O., Jézéquel, J.-M.: Security-driven model-based dynamic adaptation. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, pp. 205–214. ACM (2010)
25. Mouelhi, T., Traon, Y.L., Baudry, B.: Transforming and selecting functional test cases for security policy testing. In: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, ICST 2009, pp. 171–180. IEEE Computer Society (2009)
26. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
27. Na, S., Cheon, S.: Role delegation in role-based access control. In: Proceedings of the Fifth ACM Workshop on Role-based Access Control, RBAC 2000, pp. 39–44. ACM (2000)
28. Nguyen, P.H., Nain, G., Klein, J., Mouelhi, T., Le Traon, Y.: Model-driven adaptive delegation. In: Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development, Modularity: AOSD 2013, pp. 61–72. ACM (2013)
29. Nguyen, P.H., Papadakis, M., Rubab, I.: Testing delegation policy enforcement via mutation analsysis. In: Proceedings of the Workshop on Mutation Testing @ the Sixth IEEE International Conference on Software Testing, ICST 2013, pp. 61–72. IEEE (2013)
30. Rubio, D.: Pro Spring dynamic modules for OSGi service platforms (2009)
31. Sandhu, R., Park, J.: Usage control: A vision for next generation access control. In: Gorodetsky, V., Popyack, L.J., Skormin, V.A. (eds.) MMM-ACNS 2003. LNCS, vol. 2776, pp. 17–31. Springer, Heidelberg (2003)
32. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer 29(2), 38–47 (1996)
33. O. The OSGi Alliance. OSGi service platform core specification, release 4.1(2007)
34. Zhang, X., Oh, S., Sandhu, R.: PBDM: a flexible delegation model in RBAC. In: Proceedings of the Eighth ACM Symposium on Access Control Models and Technologies, SACMAT 2003, pp. 149–157. ACM (2003)

Effective Aspects: A Typed Monadic Embedding of Pointcuts and Advice

Ismael Figueroa[1,2,⋆], Nicolas Tabareau[2], and Éric Tanter[1,⋆⋆]

[1] PLEIAD Laboratory
University of Chile, Santiago, Chile
{ifiguero,etanter}@dcc.uchile.cl
http://pleiad.cl
[2] ASCOLA Group
INRIA – Nantes, France
nicolas.tabareau@inria.fr

Abstract. Aspect-oriented programming (AOP) aims to enhance modularity and reusability in software systems by offering an abstraction mechanism to deal with crosscutting concerns. However, in most general-purpose aspect languages aspects have almost unrestricted power, eventually conflicting with these goals. In this work we present Effective Aspects: a novel approach to embed the pointcut/advice model of AOP in a statically typed functional programming language like Haskell. Our work extends EffectiveAdvice, by Oliveira, Schrijvers, and Cook; which lacks quantification, and explores how to exploit the monadic setting in the full pointcut/advice model. Type soundness is guaranteed by exploiting the underlying type system, in particular phantom types and a new anti-unification type class. Aspects are first-class, can be deployed dynamically, and the pointcut language is extensible, therefore combining the flexibility of dynamically typed aspect languages with the guarantees of a static type system. Monads enables us to directly reason about computational effects both in aspects and base programs using traditional monadic techniques. Using this we extend Aldrich's notion of Open Modules with effects, and also with protected pointcut interfaces to external advising. These restrictions are enforced statically using the type system. Also, we adapt the techniques of EffectiveAdvice to reason about and enforce control flow properties. Moreover, we show how to control effect interference using the parametricity-based approach of EffectiveAdvice. However, this approach falls short when dealing with interference between multiple aspects. We propose a different approach using *monad views*, a recently developed technique for handling the monad stack. Finally, we exploit the properties of our monadic weaver to enable the modular construction of new semantics for aspect scoping and weaving. These semantics also benefit fully from the monadic reasoning mechanisms present in the language. This work brings type-based reasoning about effects for the first time in the pointcut/advice model, in a framework that is both expressive and extensible; thus allowing development of robust aspect-oriented systems as well as being a useful research tool for experimenting with new aspect semantics.

Keywords: aspect-oriented programming, monads, pointcut/advice model, type-based reasoning, modular reasoning.

⋆ Funded by a CONICYT-Chile Doctoral Scholarship.
⋆⋆ Partially funded by FONDECYT project 1110051.

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 145–192, 2014.
© Springer-Verlag Berlin Heidelberg 2014

1 Introduction

Aspect-oriented programming languages support the modular definition of crosscutting concerns through a join point model [19]. In the pointcut/advice mechanism, crosscutting is supported by means of pointcuts, which quantify over join points, in order to implicitly trigger advice [48]. Such a mechanism is typically integrated in an existing programming language by modifying the language processor, may it be the compiler (either directly or through macros), or the virtual machine. In a typed language, introducing pointcuts and advices also means extending the type system, if type soundness is to be preserved. For instance, AspectML [7] is based on a specific type system in order to safely apply advice. AspectJ [18] does not substantially extend the type system of Java and suffers from soundness issues. StrongAspectJ [8] addresses these issues with an extended type system. In both cases, proving type soundness is rather involved because a whole new type system has to be dealt with.

In functional programming, the traditional way to tackle language extensions, mostly for embedded languages, is to use monads [27]. Early work on AOP suggests a strong connection to monads. De Meuter proposed to use them to lay down the foundations of AOP [26], and Wand *et al.* used monads in their denotational semantics of pointcuts and advice [48]. Recently, Tabareau proposed a weaving algorithm that supports monads in the pointcut and advice model, which yields benefits in terms of extensibility of the aspect weaver [38], although in this work the weaver itself was not monadic but integrated internally in the system. This connection was exploited in recent preliminary work by the authors to construct an extensible monadic aspect weaver, in the context of Typed Racket [14], but the proposed monadic weaver was not fully typed because of limitations in the type system of Typed Racket.

This work proposes Effective Aspects: a lightweight, full-fledged embedding of aspects in Haskell, that is typed and monadic.[1] By *lightweight*, we mean that aspects are provided as a small standard Haskell library. The embedding is *full-fledged* because it supports dynamic deployment of first-class aspects with an extensible pointcut language—as is usually found only in dynamically typed aspect languages like AspectScheme [11] and AspectScript [45] (Sect. 3).

By *typed*, we mean that in the embedding, pointcuts, advices, and aspects are all statically typed (Sect. 4), and pointcut/advice bindings are proven to be safe (Sect. 5). Type soundness is directly derived by relying on the existing type system of Haskell (type classes [47], phantom types [21], and some recent extensions of the Glasgow Haskell Compiler). Specifically, we define a novel type class for anti-unification [32,33], which is key to define safe aspects.

Finally, because the embedding is *monadic*, we derive two notable advantages over ad hoc approaches to introducing aspects in an existing language. First, we can directly reason about aspects and effects using traditional monadic techniques. In short,

[1] This work is an extension of our paper in the 12th International Conference on Aspect-Oriented Software Development [39]. We mainly expand on using types to reason about aspect interference (Section 7). In addition, we provide a technical background about monadic programming in Haskell (Section 2). The implementation is available, with examples, at `http://pleiad.cl/EffectiveAspects`

we can generalize the interference combinators of EffectiveAdvice [28] in the context of pointcuts and advice (Sect. 6). And also we can use non-interference analysis techniques such as those from EffectiveAdvice, and from other advanced mechanisms, in particular *monad views* [35] (Sect. 7). Second, because we embed a monadic weaver, we can modularly extend the aspect language semantics. We illustrate this with several extensions and show how type-based reasoning can be applied to language extensions (Sect. 8). Sect. 9 discusses several issues related to our approach, Sect. 10 reviews related work, and Sect. 11 concludes.

2 Prelude: Overview of Monadic Programming

To make this work self-contained and to cater to readers not familiar with monads, we present a brief overview of the key concepts of monadic programming in Haskell used throughout this paper. More precisely, we introduce the state and error monad transformers, and the mechanisms of explicit and implicit lifting in the monad stack.

The reader is only expected to know basic Haskell programming and to understand the concept of type classes. As a good tutorial we suggest [20]. Readers already familiar with monadic programming can safely skip this section.

2.1 Monads Basics

Monads [27,46] are a mechanism to embed and reason about computational effects such as state, I/O, or exception-handling in purely functional languages like Haskell. Monad transformers [22] allow the modular construction of monads that combine several effects. A monad transformer is a type constructor used to create a *monad stack* where each layer represents an effect. Monadic programming in Haskell is provided by the Monad Transformers Library (known as *MTL*), which defines a set of monad transformers that can be flexibly composed together.

A monad is defined by a type constructor m and functions $\gg\!=$ (called $bind$) and $return$. At the type level a monad is a regular type constructor, although conceptually we distinguish a value of type a from a *computation in monad* m of type $m\ a$. Monads provide a uniform interface for computational effects, as specified in the $Monad$ type class:

$$
\begin{array}{l}
\textbf{class}\ Monad\ m\ \textbf{where} \\
\quad return :: a \to m\ a \\
\quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b
\end{array}
$$

Here $return$ promotes a value of type a into a computation of type $m\ a$, and $\gg\!=$ is a pipeline operator that takes a computation, extracts its value, and applies an action to produce a new computation. The precise meanings for $return$ and $\gg\!=$ are specific to each monad. The computational effect of a monad is "hidden" in the definition of $\gg\!=$, which imposes a sequential evaluation where the effect is performed at each step. To avoid cluttering caused by using $\gg\!=$ Haskell provides the **do**-notation, which directly

translates to chained applications of $\gg=$. The $x \leftarrow k$ expression binds identifier x with the value extracted from performing computation k for the rest of a **do** block.[2]

A monad transformer is defined by a type constructor t and the *lift* operation, as specified in the *MonadTrans* type class:

$$\textbf{class}\ \mathit{MonadTrans}\ t\ \textbf{where}$$
$$\quad \mathit{lift} :: m\ a \rightarrow t\ m\ a$$

The purpose of *lift* is to promote a computation from an inner layer of the monad stack, of type $m\ a$, into a computation in the monad defined by the complete stack, with type $t\ m\ a$. Each transformer t must declare in an effect-specific way how to make $t\ m$ an instance of the *Monad* class.

2.2 Plain Monadic Programming

To illustrate monadic programming we first describe the use of the state monad transformer *StateT*, denoted as $\mathbb{S}_\mathrm{T}$, whose computational effect is to thread a value with read-write access.

$$\textbf{newtype}\ \mathbb{S}_\mathrm{T}\ s\ m\ a = \mathbb{S}_\mathrm{T}\ (s \rightarrow m\ (a, s))$$
$$\mathit{eval}\mathbb{S}_\mathrm{T} :: \mathbb{S}_\mathrm{T}\ s\ m\ a \rightarrow s \rightarrow m\ a$$

A $\mathbb{S}_\mathrm{T}\ s\ m\ a$ computation is a function that takes an initial state of type s and returns a computation in the underlying monad m with a pair containing the resulting value of type a, and a potentially modified state of type s. The $\mathit{eval}\mathbb{S}_\mathrm{T}$ function evaluates a $\mathit{State}\ s\ m\ a$ computation using an initial state s and yields only the returning computation $m\ a$. In addition, functions $\mathit{get}\mathbb{S}_\mathrm{T}$ and $\mathit{put}\mathbb{S}_\mathrm{T}$ allow to retrieve and update the state inside a computation, respectively[3].

$$\mathit{get}\mathbb{S}_\mathrm{T} :: \mathit{Monad}\ m \Rightarrow \mathbb{S}_\mathrm{T}\ s\ m\ s$$
$$\mathit{get}\mathbb{S}_\mathrm{T} \quad = \mathbb{S}_\mathrm{T}\ \$\ \lambda s \rightarrow \mathit{return}\ (s, s)$$
$$\mathit{put}\mathbb{S}_\mathrm{T} :: \mathit{Monad}\ m \Rightarrow s \rightarrow \mathbb{S}_\mathrm{T}\ s\ m\ ()$$
$$\mathit{put}\mathbb{S}_\mathrm{T}\ s' = \mathbb{S}_\mathrm{T}\ \$\ \lambda_ \rightarrow \mathit{return}\ ((), s')$$

Note that both functions get the current state from some previous operation ($\gg=$ or $\mathit{eval}\mathbb{S}_\mathrm{T}$). The difference is that $\mathit{get}\mathbb{S}_\mathrm{T}$ returns this value and keeps the previous state unchanged, whereas $\mathit{put}\mathbb{S}_\mathrm{T}$ replaces the previous state with its argument.

Example Application. Consider a mutable queue of integers with operations to enqueue and dequeue its elements. To implement it we will define a monad stack M_1, which threads a list of integers using the $\mathbb{S}_\mathrm{T}$ transformer on top of the identity monad $\mathbb{I}$ (which has no computational effect). We also define $\mathit{run}M_1$, which initializes the queue with an empty list, and returns only the resulting value of a computation in M_1.

[2] $x \leftarrow k$ performs the effect in k, while **let** $x = k$ does not.

[3] Note the use of \$, here and throughout the rest of the paper, to avoid extra parentheses.

$$
\begin{array}{l}
\textbf{type}\ M_1 = \mathbb{S}_{\mathrm{T}}\ [\mathit{Int}]\ \mathbb{I} \\
\mathit{runM}_1 :: M_1\ a \to a \\
\mathit{runM}_1\ c = \mathit{run}\mathbb{I}\ \$\ \mathit{eval}\mathbb{S}_{\mathrm{T}}\ c\ [\,]
\end{array}
$$

The implementation of the queue operations using M_1 is simple, we just enqueue elements at the end of the list and dequeue elements from the beginning.

$$
\begin{array}{l}
\mathit{enqueue}_1 :: \mathit{Int} \to M_1\ () \\
\mathit{enqueue}_1\ n = \textbf{do}\ \mathit{queue} \leftarrow \mathit{get}\mathbb{S}_{\mathrm{T}} \\
\qquad\qquad\qquad\quad \mathit{put}\mathbb{S}_{\mathrm{T}}\ \$\ \mathit{queue} +\!\!+ [n] \\
\mathit{dequeue}_1 :: M_1\ \mathit{Int} \\
\mathit{dequeue}_1 = \textbf{do}\ \mathit{queue} \leftarrow \mathit{get}\mathbb{S}_{\mathrm{T}} \\
\qquad\qquad\qquad \mathit{put}\mathbb{S}_{\mathrm{T}}\ \$\ \mathit{tail}\ \mathit{queue} \\
\qquad\qquad\qquad \mathit{return}\ \$\ \mathit{head}\ \mathit{queue}
\end{array}
$$

Handling Error Scenarios. The above implementation of $\mathit{dequeue}_1$ terminates with a runtime error if it is performed on an empty queue, because tail fails when applied on an empty list. To provide an error-handling mechanism we use the error monad transformer ErrorT, denoted as $\mathbb{E}_{\mathrm{T}}$.

$$
\begin{array}{l}
\textbf{newtype}\ \mathbb{E}_{\mathrm{T}}\ e\ m\ a = \mathbb{E}_{\mathrm{T}}\ m\ (\mathit{Either}\ e\ a) \\
\mathit{run}\mathbb{E}_{\mathrm{T}} :: \mathit{Monad}\ m \Rightarrow \mathbb{E}_{\mathrm{T}}\ e\ m\ a \to m\ (\mathit{Either}\ e\ a)
\end{array}
$$

The type $\mathit{Either}\ e\ a$ represents two possible values: a $\mathit{Left}\ e$ value or a $\mathit{Right}\ a$ value. In this case the convention is that a $\mathit{Left}\ e$ value is treated as an error, while a $\mathit{Right}\ a$ value is considered a successful operation. Then, the $\mathit{throw}\mathbb{E}_{\mathrm{T}}$ and $\mathit{catch}\mathbb{E}_{\mathrm{T}}$ operations can be defined to raise and handle exceptions.

$$
\begin{array}{l}
\mathit{throw}\mathbb{E}_{\mathrm{T}} :: \mathit{Monad}\ m \Rightarrow e \to \mathbb{E}_{\mathrm{T}}\ e\ m\ a \\
\mathit{throw}\mathbb{E}_{\mathrm{T}}\ e = \mathbb{E}_{\mathrm{T}}\ \$\ \mathit{return}\ (\mathit{Left}\ e) \\
\mathit{catch}\mathbb{E}_{\mathrm{T}} :: \mathit{Monad}\ m \Rightarrow \mathbb{E}_{\mathrm{T}}\ e\ m\ a \to (e \to \mathbb{E}_{\mathrm{T}}\ e\ m\ a) \to \mathbb{E}_{\mathrm{T}}\ e\ m\ a \\
m\ \text{`}\mathit{catch}\mathbb{E}_{\mathrm{T}}\text{`}\ h = \mathbb{E}_{\mathrm{T}}\ \$\ \textbf{do}\ a \leftarrow \mathit{run}\mathbb{E}_{\mathrm{T}}\ m \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{case}\ a\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Left}\ \mathit{err}\ \ \to \mathit{run}\mathbb{E}_{\mathrm{T}}\ (h\ \mathit{err}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Right}\ \mathit{val} \to \mathit{return}\ (\mathit{Right}\ \mathit{val})
\end{array}
$$

Observe that $\mathit{catch}\mathbb{E}_{\mathrm{T}}$ is intended to be used as an infix operator, where the first argument is the protected expression that would be inside a *try* block in Java, while the second argument is the exception handler.

Combining State and Error-Handling Effects. To implement a queue with support for exceptions we first define a new monad stack M_2 that combines both effects (using Strings as error messages):

$$
\begin{array}{l}
\textbf{type}\ M_2 = \mathbb{S}_{\mathrm{T}}\ [\mathit{Int}]\ (\mathbb{E}_{\mathrm{T}}\ \mathit{String}\ \mathbb{I}) \\
\mathit{runM}_2\ c = \mathit{run}\mathbb{I}\ \$\ \mathit{run}\mathbb{E}_{\mathrm{T}}\ \$\ \mathit{eval}\mathbb{S}_{\mathrm{T}}\ c\ [\,]
\end{array}
$$

Then we define the $enqueue_2$ operation as before, but using M_2:

$$
\begin{aligned}
&enqueue_2 :: Int \rightarrow M_2\ ()\\
&enqueue_2\ n = \mathbf{do}\ queue \leftarrow get\mathbb{S}_\mathrm{T}\\
&\qquad\qquad\qquad put\mathbb{S}_\mathrm{T}\ \$\ queue +\!\!+ [n]
\end{aligned}
$$

However, the straightforward definition of $dequeue_2$ fails with a typing error:

```
dequeue2 :: M2 Int
dequeue2 = do queue ← getST
              if null queue
                then throwET "Queue is empty"  -- typing error
                else do putST $ tail queue
                        return $ head queue
```

The problem is that $throw\mathbb{E}_\mathrm{T}$ returns a computation whose type is $(\mathbb{E}_\mathrm{T}\ String\ \mathbb{I})\ Int$, but the return type of $dequeue_2$ is $(\mathbb{S}_\mathrm{T}\ [Int]\ (\mathbb{E}_\mathrm{T}\ String\ \mathbb{I}))\ Int$.

Explicit Lifting in the Monad Stack. Using *lift* we can reuse a function intended for an inner layer on the stack, like $throw\mathbb{E}_\mathrm{T}$. The number of *lift* calls corresponds to the distance between the top of the stack and the inner layer of the stack. Hence for $dequeue_2$ we need only one call to *lift*:

```
dequeue2 :: M2 Int
dequeue2 = do queue ← getST
              if null queue
                then (lift ∘ throwET) "Queue is empty"
                else do putST $ tail queue
                        return $ head queue
```

Although we managed to implement a queue with support for both effects, this is not satisfactory from a software engineering point of view. The reason is that plain monadic programming and explicit liftings produce a strong coupling between functions and particular monad stacks, hampering reusability and maintainability of the software.

2.3 Polymorphism on the Monad Stack

To address the coupling of functions with particular monad stacks and to expand the notion of monads as a uniform interface for computational effects, the MTL defines a set of type classes associated to particular effects. This way, monadic functions can impose constraints in the monad stack using these type classes instead of relying on a specific stack. These class constraints can be seen as *families of monads*, making a function polymorphic with respect to the concrete monadic stack used to evaluate it.

State Operations. The *MonadState* type class, denoted as $\mathbb{S}_\mathrm{M}$, defines the interface for state-related operations, and $\mathbb{S}_\mathrm{T}$ is the canonical instance of this class.[4]

[4] Expression $m \rightarrow s$ denotes a *functional dependency* [17], which means that the type of m determines the type of s, allowing a more precise control of type inference.

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathbb{S}_\mathrm{M}\ s\ m \mid m \to s \textbf{ where}\\
&\quad \mathit{get} :: m\ s\\
&\quad \mathit{put} :: s \to m\ ()
\end{aligned}
$$

Error-Handling Operations. The *MonadError* type class, denoted as $\mathbb{E}_\mathrm{M}$, defines the standard interface for error-handling operations, with $\mathbb{E}_\mathrm{T}$ as its canonical instance.

$$
\begin{aligned}
&\textbf{class } \mathit{Monad}\ m \Rightarrow \mathbb{E}_\mathrm{M}\ e\ m \mid m \to e \textbf{ where}\\
&\quad \mathit{throwError} :: e \to m\ a\\
&\quad \mathit{catchError} :: m\ a \to (e \to m\ a) \to m\ a
\end{aligned}
$$

Implicit Lifting in the Monad Stack. Going back to our example of the integer queue, the implementation using class contraints now is as follows:

```
enqueue :: (Monad m, S_M [Int] m) => Int -> m ()
enqueue n = do queue <- get
               put $ queue ++ [n]
dequeue :: (Monad m, S_M [Int] m, E_M String m) => m Int
dequeue = do queue <- get
             if null queue
                then throwError "Queue is empty"
                else do put $ tail queue
                        return $ head queue
```

Observe that the functions are defined in terms of an abstract monad m, which is required to be an instance of $\mathbb{S}_\mathrm{M}$, for insertions; and both $\mathbb{S}_\mathrm{M}$ and $\mathbb{E}_\mathrm{M}$ for retrieving values. Also note that *lift* is not required to use *throwError* in *dequeue*. The reason is that using type classes, like $\mathbb{S}_\mathrm{M}$ or $\mathbb{E}_\mathrm{M}$, an operation is automatically routed to *the first layer of the monad stack that is instance of the respective class*. The MTL defines implicit liftings between its transformers, by defining several class instances for each of them. Because of this, M_2 is instance of both $\mathbb{S}_\mathrm{M}$ and $\mathbb{E}_\mathrm{M}$.

The major limitation of implicit liftings is that it *only* chooses the first layer of a given effect. Consequently, when more than one instance of the same effect are used, *e.g.* two state transformers to hold the state of a queue and a stack, the parts of the program that access inner layers must use explicit lifting.

Explicit and implicit lifting are the standard mechanism in Haskell to handle the monad stack. The mechanism used to handle the monad stack directly determines the expressiveness of the type-based reasoning techniques, and other properties like modularity and reusability of components. This is discussed in detail in Sect. 6 and 7; in particular we show that the standard mechanism falls short to deal with interference of multiple aspects. Then we use monad views, a recent mechanism for managing the monad stack developed by Schrijvers and Oliveira [35], to propose another approach to address this situation.

3 Introducing Aspects

The fundamental premise for aspect-oriented programming in functional languages is that function applications need to be subject to aspect weaving. We introduce the term *open application* to refer to a function application that generates a join point, and consequently, can be woven.

Open Function Applications. Opening all function applications in a program or only a few selected ones is both a language design question and an implementation question. At the design level, this is the grand debate about *obliviousness* in aspect-oriented programming. Opening all applications is more flexible, but can lead to fragile aspects and unwanted encapsulation breaches. At the implementation level, opening all function applications requires either a preprocessor or runtime support.

For now, we focus on *quantification*—through pointcuts—and opt for a conservative design in which open applications are realized *explicitly* using the $\#$ operator: $f \# 2$ is the same as $f\ 2$, except that the application generates a join point that is subject to aspect weaving. We will come back to obliviousness in Sect. 9.3, showing how different answers can be provided within the context of our proposal.

Monadic Setting. Our approach to introduce aspects in a pure functional programming language like Haskell can be realized without considering effects. Nevertheless, most interesting applications of aspects rely on computational effects (*e.g.*, tracing, memoization, exception handling, etc.). We therefore adopt a monadic setting from the start. Also, as we show in Sect. 6 and 7, this allows us to exploit the approach of EffectiveAdvice [28] and other monadic reasoning mechanisms in order to perform type-based reasoning about effects in presence of aspects.

Illustration. As a basic example, recall the $enqueue$ function (Sect. 2.3) and consider the $uniqueAdv$ advice, which enforces that the argument is only passed to $proceed$ if it is not already present in the underlying list l (*e.g.*, to avoid repeated elements when representing a set using a list);

$$
\begin{aligned}
uniqueAdv\ proceed\ arg = \mathbf{do}\ & l \leftarrow get \\
& \mathbf{if}\ elem\ arg\ l \\
& \quad \mathbf{then}\ return\ () \\
& \quad \mathbf{else}\ proceed\ arg
\end{aligned}
$$

Then, in $program$ we $deploy$ an $aspect$ that reacts to applications of $enqueue$. This is specified using the pointcut $pcCall\ enqueue$.

$$
\begin{aligned}
program\ n\ m = \mathbf{do}\ & deploy\ (aspect\ (pcCall\ enqueue)\ uniqueAdv) \\
& enqueue \# n \\
& enqueue \# m \\
& showQueue
\end{aligned}
$$

Evaluating $program\ 1\ 2$ returns a string representation "[1,2]" with both elements, whereas $program\ 1\ 1$ returns "[1]" with only one element. Indeed, both results are as expected. As shown in this example, aspects consist of a pointcut/advice pair and are created with $aspect$, and deployed with $deploy$.

Our development of AOP simply relies on defining aspects (pointcuts, advices), the underlying aspect environment together with the operations to deploy and undeploy aspects, and open function application. The remainder of this section briefly presents these elements, and the following section concentrates on the main challenge: properly typing pointcuts and ensuring type soundness of pointcut/advice bindings.

3.1 Join Point Model

The support for crosscutting provided by an aspect-oriented programming language lies in its *join point model* [24]. A join point model is composed by three elements: *join points* that represents the (dynamic) steps in the execution of a program that aspects can affect, a *means of identifying* join points—here, pointcuts—and a *means of effecting* at join points—here, advices.

Join Points. Join points are function applications. A join point JP contains a function of type $a \to m\ b$, and an argument of type a. The monad m denotes the underlying computational effect stack. Note that this means that only functions that are properly lifted to a monadic context can be advised. In addition, in order for pointcuts to be able to reason about the type of advised functions, we require the functions to be $PolyTypeable$[5].

$$\mathbf{data}\ JP\ m\ a\ b = (Monad\ m, PolyTypeable\ (a \to m\ b)) \Rightarrow JP\ (a \to m\ b)\ a$$

From now on, we omit the type constraints related to $PolyTypeable$ (the $PolyTypeable$ constraint on a type is required each time the type has to be inspected dynamically; exact occurrences of this constraint can be found in the implementation).

Pointcuts. A pointcut is a predicate on the current join point. It is used to identify join points of interest. A pointcut simply returns a boolean to indicate whether it matches the given join point.

$$\mathbf{data}\ PC\ m\ a\ b = Monad\ m \Rightarrow PC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \to m\ Bool))$$

A pointcut is represented as a value of type $PC\ m\ a\ b$. Types a and b are used to ensure type safety, as discussed in Sect. 4.1. The predicate itself is a function with type $\forall a'\ b'.m\ (JP\ m\ a'\ b' \to m\ Bool)$, meaning it has access to the monad stack. The $\forall$ declaration quantifies on type variables a' and b' (using rank-2 types) because a pointcut should be able to match against any join point, regardless of the specific types involved (we come back to this in Sect. 4.1).

[5] Haskell provides the *Typeable* class to introspect monomorphic types. $PolyTypeable$ is an extension that supports both monomorphic and polymorphic types.

Pointcut Language. We provide two basic pointcut designators, $pcCall$ and $pcType$, as well as logical pointcut combinators, $pcOr$, $pcAnd$, and $pcNot$. A pointcut $pcType\ f$ matches all open applications to functions that have a type compatible with f (see Sect. 4.1 for a precise definition), and a pointcut $pcCall\ f$ matches all open applications to f.

$$
\begin{aligned}
&pcType\ f = PC\ (typePred\ (polyTypeOf\ f))\\
&\quad \mathbf{where}\ typePred\ t \ \ = return\ \$\ \lambda jp \rightarrow return\ (compareType\ t\ jp)\\
&pcCall\ f = PC\ (callPred\ f\ (polyTypeOf\ f))\\
&\quad \mathbf{where}\ callPred\ f\ t = return\ \$\ \lambda jp \rightarrow return\ (compareFun\ \ f\ jp \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad compareType\ \ t\ jp)
\end{aligned}
$$

In both cases we use the $polyTypeOf$ function (provided by $PolyTypeable$) to obtain the type representation of function f, and compare it to the type of the function in the join point using $compareType$. Additionally, to implement $pcCall$ we require a notion of function equality[6]. This is used in $compareFun$ to compare the function in the join point with the given function f. Note that in $pcCall$ we also need to perform a type comparison, using $compareType$. This is because a polymorphic function whose type variables are instantiated in one way is equal to the same function but with type variables instantiated in some other way (*e.g.* $id :: Int \rightarrow Int$ is equal to $id :: Float \rightarrow Float$).

Users can define their own pointcut designators. For instance, we can define control-flow pointcuts like AspectJ's *cflow* (described in Sect. 8.1), data flow pointcuts [23], pointcuts that rely on the trace of execution [9] (Sect. 7.1), etc.

Advice. An advice is a function that executes in place of a join point matched by a pointcut. This replacement is similar to open recursion in EffectiveAdvice [28]. An advice receives a function (known as the *proceed* function) and returns a new function of the same type (which may or may not apply the original *proceed* function internally). We introduce a type alias for advice:

$$\mathbf{type}\ Advice\ m\ a\ b = (a \rightarrow m\ b) \rightarrow a \rightarrow m\ b$$

For instance, the type $Monad\ \ m\ \Rightarrow\ Advice\ \ m\ \ Int\ \ Int$ is a synonym for the type $Monad\ \ m\ \Rightarrow\ (Int\ \rightarrow\ m\ \ Int)\ \rightarrow\ Int\ \rightarrow\ m\ \ Int$. For a given advice of type $Advice\ m\ a\ b$, we call $a \rightarrow m\ b$ the *advised type* of the advice.

Aspects. An aspect is a first-class value binding together a pointcut and an advice. Supporting first-class aspects is important: it makes it possible to support aspect factories, separate creation and deployment/undeployment of aspects, exporting opaque, self-contained aspects as single units, etc. We introduce a data definition for aspects, parameterized by a monad m (which has to be the same in the pointcut and advice):

$$\mathbf{data}\ Aspect\ m\ a\ b\ c\ d = Aspect\ (PC\ m\ a\ b)\ (Advice\ m\ c\ d)$$

We defer the detailed definition of $Aspect$ with its type class constraints to Sect. 4.2, when we address the issue of safe pointcut/advice binding.

[6] For this notion of function equality, we use the $StableNames$ API, which relies on pointer comparison. See Sect. 9.1 for discussion on the issues of this approach.

3.2 Aspect Deployment

Aspect Environment. The list of aspects that are deployed at a given point of time is known as the *aspect environment*. To be able to define the type $AspectEnv$ as an heterogenous list of aspects, we use an existentially quantified[7], data $EAspect$ that hides the type parameters of $Aspect$:[8]

$$\textbf{data}\ EAspect\ m = \forall a\ b\ c\ d. EAspect\ (Aspect\ m\ a\ b\ c\ d)$$
$$\textbf{type}\ AspectEnv\ m = [EAspect\ m]$$

This environment can be either fixed initially and used globally [24], as in AspectJ, or it can be handled dynamically, as in AspectScheme [11]. Different scoping strategies are possible when dealing with dynamic deployment [40]. Because we are in a monadic setting, we can pass the aspect environment implicitly using a monad. An open function application can then trigger the set of currently deployed aspects by retrieving these aspects from the underlying monad.

There are a number of design options for the aspect environment, depending on the kind of aspect deployment that is desired. Following the *Reader* monad, we can provide a fixed aspect environment, and add the ability to deploy an aspect for the dynamic extent of an expression, similarly to the *local* method of the *Reader* monad. We can also adopt a state-like monad, in order to support dynamic aspect deployment and undeployment with global scope. In this paper, without loss of generality, we go for the latter.

The $\mathbb{A}_T$ Monad Transformer. Because we are interested in using arbitrary computational effects in programs, we define the aspect environment through a *monad transformer* (Sect. 2.1), which allows the programmer to construct a monad stack of effects. The $\mathbb{A}_T$ monad transformer is defined as follows:

$$\textbf{newtype}\ \mathbb{A}_T\ m\ a = \mathbb{A}_T\ (\mathbb{S}_T\ (AspectEnv\ (\mathbb{A}_T\ m))\ m\ a)\ \textbf{deriving}\ (Monad)$$

To define the $\mathbb{A}_T$ transformer we reuse the $\mathbb{S}_T$ data constructor, because the $\mathbb{A}_T$ transformer is essentially a state transformer (Sect. 2.2) that threads the aspect environment. Using the $GeneralizedNewtypeDeriving$ extension of GHC, we can automatically derive $\mathbb{A}_T$ as an instance of $Monad$. We also define a proper instance of $MonadTrans$ (not shown here), and implicit liftings for the standard monad transformers of the MTL.[9] Observe that the aspect environment is bound to the same monad $\mathbb{A}_T\ m$, in order to provide aspects with access to open applications.

---

[7] In Haskell an existentially quantified data type is declared using $\forall$ before the data constructor.

[8] Because we cannot anticipate a fixed set of class constraints for deployed aspects, we left the type parameters unconstrained. Aspects with ad hoc polymorphism have to be instantiated before deployment to statically solve each remaining type class constraint (see Sect. 9.2 for more details).

[9] In the rest of the paper we use the same technique to define our custom monad transformers, hence we omit the **deriving** clauses and standard instance definitions, like $MonadTrans$.

Dynamic Aspect Deployment. We now define the functions for dynamic deployment, which simply add and remove an aspect from the aspect environment:

$$
\begin{aligned}
&\mathit{deploy}, \mathit{undeploy} :: \mathit{EAspect}\ (\mathbb{A}_{\mathrm{T}}\ m) \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ ()\\
&\mathit{deploy}\ \mathit{asp} \quad\ = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \mathit{aenv} \rightarrow \mathit{return}\ ((), \mathit{asp} : \mathit{aenv})\\
&\mathit{undeploy}\ \mathit{asp} = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \mathit{aenv} \rightarrow \mathit{return}\ ((), \mathit{deleteAsp}\ \mathit{asp}\ \mathit{aenv})
\end{aligned}
$$

Finally, in order to extract the computation of the underlying monad from an $\mathbb{A}_{\mathrm{T}}$ computation we define the $run\mathbb{A}_{\mathrm{T}}$ function, with type $\mathit{Monad}\ m \Rightarrow \mathbb{A}_{\mathrm{T}}\ m\ a \rightarrow m\ a$ (similar to $eval\mathbb{S}_{\mathrm{T}}$ in the state monad transformer), that runs a computation in an empty initial aspect environment. For instance, in the initial example of the $\mathit{enqueue}$ function, we can define a client as follows:

$$
\mathit{client}\ n\ m = \mathit{run}\mathbb{I}\ (\mathit{run}\mathbb{A}_{\mathrm{T}}\ (\mathit{program}\ n\ m))
$$

3.3 Aspect Weaving

Aspect weaving is triggered through open applications, *i.e.*, applications performed with the # operator, *e.g.*, $f \# x$.

Open Applications. We introduce a type class $\mathit{OpenApp}$ that declares the # operator. This makes it possible to overload # in certain contexts, and it can be used to declare constraints on monads to ensure that the operation is available in a given context.

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{OpenApp}\ m\ \mathbf{where}\\
&\quad (\#) :: (a \rightarrow m\ b) \rightarrow a \rightarrow m\ b
\end{aligned}
$$

The # operator takes a function of type $a \rightarrow m\ b$ and returns a (woven) function with the same type. Any monad composed with the $\mathbb{A}_{\mathrm{T}}$ transformer has open application defined:

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{OpenApp}\ (\mathbb{A}_{\mathrm{T}}\ m)\ \mathbf{where}\\
&\quad f \# a = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \mathit{aenv} \rightarrow \mathbf{do}\\
&\qquad\qquad (\mathit{woven_f}, \mathit{aenv}') \leftarrow \mathit{weave}\ f\ \mathit{aenv}\ \mathit{aenv}\ (\mathit{newjp}\ f\ a)\\
&\qquad\qquad \mathit{run}\ (\mathit{woven_f}\ a)\ \mathit{aenv}'
\end{aligned}
$$

An open application results in the creation of a join point, $\mathit{newjp}\ f\ a$, that represents the application of f to a. The join point is then used to determine which aspects in the environment match, produce a new function that combines all the applicable advices, and apply that function to the original argument.

Weaving. The function to use at a given point is produced by the weave function:

$$
\begin{aligned}
&\mathit{weave} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ b) \rightarrow \mathit{AspectEnv}\ (\mathbb{A}_{\mathrm{T}}\ m) \rightarrow\\
&\quad \mathit{AspectEnv}\ (\mathbb{A}_{\mathrm{T}}\ m) \rightarrow \mathit{JP}\ (\mathbb{A}_{\mathrm{T}}\ m)\ a\ b \rightarrow m\ (a \rightarrow \mathbb{A}_{\mathrm{T}}\ m\ b, \mathit{AspectEnv}\ (\mathbb{A}_{\mathrm{T}}\ m))\\
&\mathit{weave}\ f\ []\ \mathit{fenv}\ _ \qquad\qquad = \mathit{return}\ (f, \mathit{fenv})\\
&\mathit{weave}\ f\ (\mathit{asp} : \mathit{asps})\ \mathit{fenv}\ \mathit{jp} =\\
&\quad \mathbf{case}\ \mathit{asp}\ \mathbf{of}\ \mathit{EAspect}\ (\mathit{Aspect}\ \mathit{pc}\ \mathit{adv}) \rightarrow
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{do}\ (match, fenv') \leftarrow apply_pc\ pc\ jp\ fenv \\
&\quad weave\ (\textbf{if}\ match \\
&\qquad\qquad \textbf{then}\ apply_adv\ adv\ f \\
&\qquad\qquad \textbf{else}\ f) \\
&\qquad asps\ fenv'\ jp
\end{aligned}
$$

The $weave$ function is defined recursively on the aspect environment. For each aspect, it applies the pointcut to the join point. It then uses either the partial application of the advice to f if the pointcut matches, or f otherwise[10], to keep on weaving on the rest of the aspect list. This definition is a direct adaptation of AspectScheme's weaving function [11], and is also a *monadic weaver* [38] that supports modular language extensions (in Sect. 8 we show how to exploit this feature).

Applying Advice. As we have seen, the aspect environment has type $AspectEnv\ m$, meaning that the type of the advice function is hidden. Therefore, advice application requires *coercing* the advice to the proper type in order to apply it to the function of the join point:

$$
\begin{aligned}
&apply_adv :: Advice\ m\ a\ b \rightarrow t \rightarrow t \\
&apply_adv\ adv\ f = (unsafeCoerce\ adv)\ f
\end{aligned}
$$

The operation $unsafeCoerce$ of Haskell is (unsurprisingly) unsafe and can yield segmentation faults or arbitrary results. To recover safety, we could insert a runtime type check with $compareType$ just before the coercion. We instead make aspects type safe such that we can prove that the particular use of $unsafeCoerce$ in $apply_adv$ is *always* safe. The following section describes how we achieve type soundness of aspects; Sect. 5 formally proves it.

4 Typing Aspects

Ensuring type soundness in the presence of aspects consists in ensuring that an advice is always applied at a join point of the proper type. Note that by “the type of the join point,” we refer to the type of the function being applied at the considered join point.

4.1 Typing Pointcuts

The intermediary between a join point and an advice is the pointcut, whose proper typing is therefore crucial. The type of a pointcut as a predicate over join points does not convey any information about the types of join points it matches. To keep this information, we use *phantom type variables* a and b in the definition of PC:

$$\textbf{data}\ PC\ m\ a\ b = Monad\ m \Rightarrow PC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \rightarrow m\ Bool))$$

[10] $apply_pc$ checks whether the pointcut matches the join point and returns a boolean and a potentially modified aspect environment. Note that $apply_pc$ is evaluated in the full aspect environment $fenv$, instead of the decreasing $(asp : asps)$ argument.

A phantom-type variable is a type variable that is not used on the right-hand side of the data-type definition. The use of phantom type variables to type embedded languages was first introduced by Leijen and Meijer to type an embedding of SQL in Haskell [21]; it makes it possible to "tag" extra type information on data. In our context, we use it to add the information about the type of the join points matched by a pointcut: $PC\ m\ a\ b$ means that a pointcut can match applications of functions of type $a \to m\ b$. We call this type the *matched type* of the pointcut. Pointcut designators are in charge of specifying the matched type of the pointcuts they produce.

Least General Types. Because a pointcut potentially matches many join points of different types, the matched type must be a *more general type*. For instance, consider a pointcut that matches applications of functions of type $Int \to m\ Int$ and $Float \to m\ Int$. Its matched type is the parametric type $a \to m\ Int$. Note that this is in fact the *least general type* of both types.[11] Another more general candidate is $a \to m\ b$, but the least general type conveys more precise information. As a concrete example, below is the type signature of the $pcCall$ pointcut designator:

$$pcCall :: Monad\ m \Rightarrow (a \to m\ b) \to PC\ m\ a\ b$$

Comparing Types. The type signature of the $pcType$ pointcut designator is the same as that of $pcCall$:

$$pcType :: Monad\ m \Rightarrow (a \to m\ b) \to PC\ m\ a\ b$$

However, suppose that f is a function of type $Int \to m\ a$. We want the pointcut $pcType\ f$ to match applications of functions of more specific types, such as $Int \to m\ Int$ and $Int \to m\ Char$. This means that $compareType$ actually checks that the matched type of the pointcut is *more general* than the type of the join point.

Logical Combinators. We use type constraints in order to properly specify the matched type of logical combinations of pointcuts. The intersection of two pointcuts matches join points that are most precisely described by the *principal unifier* of both matched types. Since Haskell supports this unification when the same type variable is used, we can simply define $pcAnd$ as follows:

$$pcAnd :: Monad\ m \Rightarrow PC\ m\ a\ b \to PC\ m\ a\ b \to PC\ m\ a\ b$$

For instance, a control flow pointcut matches any type of join point, so its matched type is $a \to m\ b$. Consequently, if f is of type $Int \to m\ a$, the matched type of $pcAnd\ (pcCall\ f)\ (pcCflow\ g)$ is $Int \to m\ a$.

Dually, the union of two pointcuts relies on *anti-unification* [32,33], that is, the computation of the least general type of two types. Haskell does not natively support anti-unification. We exploit the fact that multi-parameter type classes can be used to

[11] The term *most specific generalization* is also valid, but we stick here to Plotkin's original terminology [32].

define relations over types, and develop a novel type class $LeastGen$ (for *least general*) that can be used as a constraint to compute the least general type t of two types t_1 and t_2 (defined in Sect. 5):

$$\begin{aligned} &pcOr :: (Monad\ m, LeastGen\ (a \to b)\ (c \to d)\ (e \to f)) \Rightarrow \\ &\quad PC\ m\ a\ b \to PC\ m\ c\ d \to PC\ m\ e\ f \end{aligned}$$

For instance, if f is of type $Int \to m\ a$ and g is of type $Int \to m\ Float$, the matched type of $pcOr\ (pcCall\ f)\ (pcCall\ g)$ is $Int \to m\ a$.

The negation of a pointcut can match join points of any type because no assumption can be made on the matched join points:

$$pcNot :: Monad\ m \Rightarrow PC\ m\ a\ b \to PC\ m\ a'\ b'$$

Observe that the type of $pcNot$ is quite restrictive. In fact, the advice of any aspect with a single $pcNot$ pointcut must be completely generic because the matched type corresponds to fresh type variables. The matched type of $pcNot$ can be made more specific using $pcAnd$ to combine it with other pointcuts with more specific types.

Open Pointcut Language. The set of pointcut designators in our language is open. User-defined pointcut designators are, however, responsible for properly specifying their matched types. If the matched type is incorrect or too specific, soundness is lost.

Constraining Pointcuts to Specific Types. A pointcut cannot make any type assumption about the type of the join point it receives as argument. The reason for this is again the homogeneity of the aspect environment: when deploying an aspect, the type of its pointcut is hidden. At runtime, then, a pointcut is expected to be applicable to any join point. The general approach to make a pointcut safe is therefore to perform a runtime-type check, as was illustrated in the definition of $pcCall$ and $pcType$ in Sect. 3.1. However, certain pointcuts are meant to be conjoined with others pointcuts that will first apply a sufficient type condition.

In order to support the definition of pointcuts that *require* join points to be of a given type, we provide the $RequirePC$ type:

$$\begin{aligned} &\mathbf{data}\ RequirePC\ m\ a\ b = Monad\ m \Rightarrow \\ &\qquad RequirePC\ (\forall a'\ b'.m\ (JP\ m\ a'\ b' \to m\ Bool)) \end{aligned}$$

The definition of $RequirePC$ is similar to that of PC, with two important differences. First, the matched type of a $RequirePC$ is interpreted as a type *requirement*. Second, a $RequirePC$ is not a valid stand-alone pointcut: it has to be combined with a standard PC that enforces the proper type upfront. To safely achieve this, we overload $pcAnd$[12]:

$$\begin{aligned} &pcAnd :: (Monad\ m, LessGen\ (a \to b)\ (c \to d)) \Rightarrow \\ &\qquad PC\ m\ a\ b \to RequirePC\ m\ c\ d \to PC\ m\ a\ b \end{aligned}$$

[12] The constraint is different from the previous constraint on $pcAnd$. This is possible thanks to the recent $ConstraintKinds$ extension of GHC.

In this case $pcAnd$ yields a standard PC pointcut and checks that the matched type of the PC pointcut is *less general* than the type expected by the $RequirePC$ pointcut. This is expressed using the constraint $LessGen$, which, as we will see in Sect. 5, is based on $LeastGen$.

To illustrate, let us define a pointcut designator $pcArgGT$ for specifying pointcuts that match when the argument at the join point is greater than a given n (of type a instance of the Ord type class):

$$
\begin{array}{l}
pcArgGT :: (Monad\ m, Ord\ a) \Rightarrow a \rightarrow RequirePC\ m\ a\ b \\
pcArgGT\ n = RequirePC\ \$\ return\ (\lambda jp \rightarrow \\
\quad return\ (unsafeCoerce\ (getJpArg\ jp) \geqslant n))
\end{array}
$$

The use of $unsafeCoerce$ to coerce the join point argument to the type a forces us to declare the Ord constraint on a when typing the returned pointcut as $RequirePC\ m\ a\ b$ (with a fresh type variable b). To get a proper pointcut, we use $pcAnd$, for instance to match all calls to $enqueue$ where the argument is greater than 10:

$$pcCall\ enqueue\ `pcAnd`\ pcArgGT\ 10$$

The $pcAnd$ combinator guarantees that a $pcArgGT$ pointcut is always applied to a join point with an argument that is indeed of a proper type: no runtime type check is necessary within $pcArgGT$, because the coercion is always safe.

4.2 Typing Aspects

The main typing issue we have to address consists in ensuring that a pointcut/advice binding is type safe, so that the advice application does not fail. A first idea to ensure that the pointcut/advice binding is type safe is to require the matched type of the pointcut and the advised type of the advice to be the same (or rather, unifiable):

$$
\begin{array}{l}
\quad \text{-- wrong!} \\
\mathbf{data}\ Aspect\ m\ a\ b = Aspect\ (PC\ m\ a\ b)\ (Advice\ m\ a\ b)
\end{array}
$$

This approach can however yield unexpected behavior. Consider this example:

$$
\begin{array}{l}
idM\ x = return\ x \\
adv :: Monad\ m \Rightarrow Advice\ (Char \rightarrow m\ Char) \\
adv\ proceed\ c = proceed\ (toUpper\ c) \\
program = \mathbf{do}\ deploy\ (aspect\ (pcCall\ idM)\ adv) \\
\qquad\qquad\quad x \leftarrow idM\ \#\ \texttt{'a'} \\
\qquad\qquad\quad y \leftarrow idM\ \#\ [True, False, True] \\
\qquad\qquad\quad return\ (x, y)
\end{array}
$$

The matched type of the pointcut $pcCall\ idM$ is $Monad\ m \Rightarrow a \rightarrow m\ a$. With the above definition of $Aspect$, $program$ passes the typechecker because it is possible to unify a and $Char$ to $Char$. However, when evaluated, the behavior of $program$ is undefined because the advice is unsafely applied with an argument of type $[Bool]$, for which $toUpper$ is undefined.

The problem is that during typechecking, the matched type of the pointcut and the advised type of the advice can be unified. Because unification is symmetric, this succeeds even if the advised type is more specific than the matched type. In order to address this, we again use the type class $LessGen$ to ensure that the matched type is less general than the advice type:

$$\textbf{data}\ Aspect\ m\ a\ b\ c\ d = (Monad\ m, LessGen\ (a \to m\ b)\ (c \to m\ d)) \Rightarrow \\ Aspect\ (PC\ m\ a\ b)\ (Advice\ m\ c\ d)$$

This constraint ensures that pointcut/advice bindings are type safe: the coercion performed in $apply_adv$ always succeeds. We formally prove this in the following section.

5 Typing Aspects, Formally

We now formally prove the safety of our approach. We start briefly summarizing the notion of type substitutions and the *is less general* relation between types. Note that we do not consider type class constraints in the definition. Then we describe a novel anti-unification algorithm implemented with type classes, on which the type classes $LessGen$ and $LeastGen$ are based. We finally prove pointcut and aspect safety, and state our main safety theorem.

5.1 Type Substitutions

In this section, we summarize the definition of type substitutions and introduce formally the notion of least general type in a Haskell-like type system (without ad-hoc polymorphism). Thus, we have types $t ::= Int, Char, \ldots, t_1 \to t_2, T\ t_1\ \ldots\ t_m$, which denote primitive types, functions, and m-ary type constructors, in addition to user-defined types. We consider a typing environment $\Gamma = (x_i : t_i)_{i \in \mathbb{N}}$ that binds variables to types.

Definition 1 (Type Substitution, from [31]). *A type substitution σ is a finite mapping from type variables to types. It is denoted $[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$, where $dom(\sigma)$ and $range(\sigma)$ are the sets of types appearing in the left-hand and right-hand sides of the mapping, respectively. It is possible for type variables to appear in $range(\sigma)$.*

Substitutions are always applied simultaneously on a type. If σ and γ are substitutions, and t is a type, then $\sigma \circ \gamma$ is the composed substitution, where $(\sigma \circ \gamma)t = \sigma(\gamma t)$. Application of substitution on a type is defined inductively on the structure of the type.

Substitution is extended pointwise for typing environments in the following way: $\sigma(x_i : t_i)_{i \in \mathbb{N}} = (x_i : \sigma t_i)_{i \in \mathbb{N}}$. Also, applying a substitution to a term t means to apply the substitution to all type annotations appearing in t.

Definition 2 (Less General Type). *We say type t_1* is less general *than type t_2, denoted $t_1 \preceq t_2$, if there exists a substitution σ such that $\sigma t_2 = t_1$. Observe that $\preceq$ defines a partial order on types (modulo α-renaming).*

Definition 3 (Least General Type). *Given types t_1 and t_2, we say type t is the* least general type *iff t is the supremum of t_1 and t_2 with respect to $\preceq$.*

```
1  class LeastGen' a b c σ_in σ_out | a b c σ_in → σ_out
2    -- Inductive case: The two type constructors match,
3    -- recursively compute the substitution for type arguments a_i,b_i.
4  instance (LeastGen' a_1 b_1 c_1 σ_0 σ_1, ...,
5            LeastGen' a_n b_n c_n σ_{n-1} σ_n,
6            T c_1 ... c_n ~ c)
7    ⇒ LeastGen' (T a_1 ... a_n) (T b_1 ... b_n) c σ_0 σ_n
8    -- Default case: The two type constructors don't match, c has to be a variable,
9    -- either unify c with c' if c' ↦ (a, b), or extend the substitution with c ↦ (a, b)
10 instance (Analyze c (TVar c),
11           MapsTo σ_in c' (a, b),
12           VarCase c' (a, b) c σ_in σ_out)
13   ⇒ LeastGen' a b c σ_in σ_out
```

Fig. 1. The $LeastGen'$ type class. An instance holds if c is the least general type of a and b.

5.2 Statically Computing Least General Types

In an aspect declaration, we statically check the type of the pointcut and the type of the advice to ensure a safe binding. To do this we encode an anti-unification algorithm at the type level, exploiting the type class mechanism. A multi-parameter type class $R\ t_1 \dots t_n$ can be seen as a *relation* R on types $t_1 \dots t_n$, and **instance** declarations as ways to inductively define this relation, in a manner very similar to logic programming.

The type classes $LessGen$ and $LeastGen$ used in Sect. 4 are defined as particular cases of the more general type class $LeastGen'$, shown in Fig. 1. This class is defined in line 1 and is parameterized by types a, b, c, σ_{in}, and σ_{out}. Note that σ_{out} is functionally dependent on a, b, c, and σ_{in}; and that there is no **where** keyword because the class declares no operations. Both σ_{in} and σ_{out} denote substitutions encoded at the type level as a list of mappings from type variables to *pairs* of types. We use pairs of types in substitutions because we have to simultaneously compute substitutions from c to a and from c to b.

To be concise, lines 4-7 present a single definition parametrized by the type constructor arity but in practice, there needs to be a different instance declaration for each type constructor arity.

Proposition 1. *If* $LeastGen'\ a\ b\ c\ \sigma_{in}\ \sigma_{out}$ *holds, then substitution* σ_{out} *extends* σ_{in} *and* $\sigma_{out} c = (a, b)$.

Proof. By induction on the type representation of a *and* b.

A type can either be a type variable, represented as $TVar\ a$, *or an* n*-ary type constructor* T *applied to* n *type arguments*[13]. *The rule to be applied depends on whether the type constructors of* a *and* b *are the same or not.*

(i) *If the constructors are the same, then the rule defined in lines 4-7 computes* $(T\ c_1 \dots c_n)$ *using the induction hypothesis that* $\sigma_i c_i = (a_i, b_i)$, *for* $i = 1 \dots n$. *The*

[13] We use the $Analyze$ type class from $PolyTypeable$ to get a type representation at the type level. For simplicity we omit the rules for analyzing type representations.

component-wise application of constraints is done from left to right, starting from substitution σ_0 and extending it to the resulting substitution σ_n. The type equality constraint $(T\ c_1 \ldots c_n) \sim c$ checks that c is unifiable with $(T\ c_1 \ldots c_n)$ and, if so, unifies them. Then, we can check that $\sigma_n c = (a, b)$.

*(*ii*) If the type constructors are not the same the only possible generalization is a type variable. In the rule defined in lines 10-13 the goal is to extend σ_{in} with the mapping $c \mapsto (a, b)$ such that $\sigma_{out} c = (a, b)$, while preserving the injectivity of the substitution (see next proposition).* □

Proposition 2. *If σ_{in} is an injective function, and $LeastGen'\ a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then σ_{out} is an injective function.*

Proof. By construction $LeastGen'$ introduces a binding from a fresh type variable to (a, b), in the rule defined in lines 10-13, only if there is no type variable already mapping to (a, b)—in which case σ_{in} is not modified.

To do this, we first check that c is actually a type variable $(TVar\ c)$ by checking its representation using $Analyze$. Then in relation $MapsTo$ we bind c' to the (possibly inexistent) type variable that maps to (a, b) in σ_{in}. In case there is no such mapping, then c' is $None$.

Finally, relation $VarCase$ binds σ_{out} to σ_{in} extended with $\{c \mapsto (a, b)\}$ in case c' is $None$, otherwise $\sigma_{out} = \sigma_{in}$. It then unifies c with c'. In all cases c is bound to the variable that maps to (a, b) in σ_{out}, because it was either unified in rule $MapsTo$ or in rule $VarCase$. The hypothesis that σ_{in} is injective ensures that any preexisting mapping is unique. □

Proposition 3. *If σ_{in} is an injective function, and $LeastGen'\ a\ b\ c\ \sigma_{in}\ \sigma_{out}$ holds, then c is the least general type of a and b.*

Proof. By induction on the type representation of a and b.

*(*i*) If the type constructors are different the only generalization possible is a type variable c.*

*(*ii*) If the type constructors are the same, then $a = T\ a_1 \ldots a_n$ and $b = T\ b_1 \ldots b_n$. By Proposition 1, $c = T\ c_1 \ldots c_n$ generalizes a and b with the substitution σ_{out}. By induction hypothesis c_i is the least general type of (a_i, b_i).*

Now consider a type d that also generalizes a and b, i.e. *$a \preceq d$ and $b \preceq d$, with associated substitution α. We prove c is less general than d by constructing a substitution τ such that $\tau d = c$.*

Again, there are two cases, either d is a type variable, in which case we set $\tau = \{d \mapsto c\}$, or it has the same outermost type constructor, i.e. *$d = T d_1 \ldots d_n$. Thus $a_i \preceq d_i$ and $b_i \preceq d_i$; and because c_i is the least general type of a_i and b_i, there exists a substitution τ_i such that $\tau_i d_i = c_i$, for $i = 1 \ldots n$.*

Now consider a type variable $x \in dom(\tau_i) \cap dom(\tau_j)$. By definition of α, we know that $\sigma_{out}(\tau_i(x)) = \alpha(x)$ and $\sigma_{out}(\tau_j(x)) = \alpha(x)$. Because σ_{out} is injective (by Proposition 2), we deduce that $\tau_i(x) = \tau_j(x)$ so there are no conflicting mappings between τ_i and τ_j, for any i and j. Consequently, we can define $\tau = \bigcup \tau_i$ and check that $\tau d = c$. □

Definition 4 (LeastGen type class). *To compute the least general type* c *for* a *and* b*, we define:*

$LeastGen\ a\ b\ c \triangleq\ LeastGen'\ a\ b\ c\ \sigma_{empty}\ \sigma_{out}$*, where* σ_{empty} *is the empty substitution and* σ_{out} *is the resulting substitution.*

Definition 5 (LessGen type class). *To establish that type* a *is less general than type* b*, we define:*

$LessGen\ a\ b \triangleq\ LeastGen\ a\ b\ b$

5.3 Pointcut Safety

We now establish the safety of pointcuts with relation to join points.

Definition 6 (Pointcut match). *We define the relation* $matches(pc, jp)$*, which holds iff applying pointcut* pc *to join point* jp *in the context of a monad* m *yields a computation* $m\ True$*.*

Definition 7 (Safe user-defined pointcut). *Given a join point term* jp *and type environment* Γ*, a user-defined pointcut is safe if:*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash jp : JP\ m\ a'\ b'$
$\Gamma \vdash matches(pc, jp)$
implies that $a' \rightarrow m\ b' \preceq a \rightarrow m\ b$*.*

Now we prove that the matched type of a given pointcut is more general than the join points matched by that pointcut.

Proposition 4. *Given a join point term* jp *and a pointcut term* pc*, and type environment* Γ*; and that if* pc *is user-defined, then it is safe (according to Definition 7). Then, if*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash jp : JP\ m\ a'\ b'$
$\Gamma \vdash matches(pc, jp)$
then $a' \rightarrow m\ b' \preceq a \rightarrow m\ b$*.*

Proof. By induction on the matched type of the pointcut.

- *Case* $pcCall$*: By construction the matched type of a* $pcCall\ f$ *pointcut is the type of* f*. Such a pointcut matches a join point with function* g *if and only if:* f *is equal to* g*, and the type of* f *is less general than the type of* g*. (On both* $pcCall$ *and* $pcType$ *this type comparison is performed by* $compareType$ *on the type representations of its arguments.)*
- *Case* $pcType$*: By construction the matched type of a* $pcType\ f$ *pointcut is the type of* f*. Such a pointcut only matches a join point with function* g *whose type is less general than the matched type.*
- *Case* $pcAnd$ *on* $PC\ PC$*: Consider* pc_1 *'*$pcAnd$*'* pc_2*. The matched type of the combined pointcut is the* principal unifier *of the matched types of the arguments—which represents the intersection of the two sets of join points. The property holds by the induction hypothesis applied to* pc_1 *and* pc_2*.*

- *Case $pcAnd$ on $PC\ RequirePC$: Consider pc_1 ‘$pcAnd$‘ pc_2. The matched type of the combined pointcut is the type of pc_1 and it is checked that the type required by pc_2 is* more general *so the application of pc_2 will not yield an error. The property holds by induction hypothesis on pc_1.*
- *Case $pcOr$: Consider pc_1 ‘$pcOr$‘ pc_2. The matched type of the combined pointcut is the* least general type *of the matched types of the argument, computed by the $LeastGen$ constraint—which represents the union of the two sets of join points. The property holds by induction hypothesis on pc_1 and pc_2.*
- *Case $pcNot$: The matched type of a pointcut constructed with $pcNot$ is a fresh type variable, which by definition is more general than the type of any join point.* □

5.4 Advice Type Safety

If an aspect is well-typed, then the advised type of the advice is more general than the matched type of the pointcut:

Proposition 5. *Given a pointcut term pc, an advice term adv, and a type environment Γ, if*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash adv : Advice\ m\ c\ d$
$\Gamma \vdash (aspect\ pc\ adv) : Aspect\ m\ a\ b\ c\ d$
then $a \to m\ b \preceq c \to m\ d$.

Proof. *Using the definition of $Aspect$ (Sect. 4.2) and because $\Gamma \vdash (aspect\ pc\ adv) : Aspect\ m\ a\ b\ c\ d$, we know that the constraint LessGen is satisfied, so by Definitions 4 and 5, and Proposition 1, $a \to m\ b \preceq c \to m\ d$.* □

5.5 Safe Aspects

We now show that if an aspect is well-typed, then the advised type of the advice is more general than the type of join points matched by the corresponding pointcut:

Theorem 1 (Safe Aspects). *Given the terms jp, pc, and adv representing a join point, a pointcut and an advice respectively, given a type environment Γ; and assuming that if pc is a user-defined pointcut, then it is safe (according to Definition 7). Then, if*
$\Gamma \vdash pc : PC\ m\ a\ b$
$\Gamma \vdash adv : Advice\ m\ c\ d$
$\Gamma \vdash (aspect\ pc\ adv) : Aspect\ m\ a\ b\ c\ d$
and
$\Gamma \vdash jp : JP\ m\ a'\ b'$
$\Gamma \vdash matches(pc, jp)$
then $a' \to m\ b' \preceq c \to m\ d$.

Proof. *By Proposition 4 and 5 and the transitivity of $\preceq$.* □

$$\begin{array}{l}
\textbf{module}\ \mathit{Fib}\ (\mathit{fib}, \mathit{pcFib})\ \textbf{where} \\
\textbf{import}\ AOP \\
\mathit{pcFib} = \mathit{pcCall}\ \mathit{fibBase}\ `\mathit{pcAnd}`\ \mathit{pcArgGT}\ 2 \\
\mathit{fibBase}\ n = \mathit{return}\ 1 \\
\mathit{fibAdv}\ \mathit{proceed}\ n = \textbf{do}\ \mathit{f1} \leftarrow \mathit{fibBase}\ \#\ (n-1) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{f2} \leftarrow \mathit{fibBase}\ \#\ (n-2) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{f1} + \mathit{f2}) \\
\mathit{fib} :: \mathit{Monad}\ m \Rightarrow m\ (\mathit{Int} \rightarrow m\ \mathit{Int}) \\
\mathit{fib} = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ \mathit{fibAdv}) \\
\qquad\qquad \mathit{return}\ \$\ \mathit{fibBase}\ \ \#
\end{array}$$

Fig. 2. Fibonacci module

Corollary 1 (Safe Weaving). *The coercion of the advice in apply_adv is safe.*

Proof. Recall apply_adv (Sect. 3.3):

$$\begin{array}{l}
\mathit{apply_adv} :: \mathit{Advice}\ m\ a\ b \rightarrow t \rightarrow t \\
\mathit{apply_adv}\ \mathit{adv}\ f = (\mathit{unsafeCoerce}\ \mathit{adv})\ f
\end{array}$$

By construction, $\mathit{apply_adv}$ *is used only with a function* f *that comes from a join point that is matched by a pointcut associated to* adv*. Using Theorem 1, we know that the join point has type* $JP\ m\ a'\ b'$ *and that* $a' \rightarrow m\ b' \preceq a \rightarrow m\ b$*. We note* σ *the associated substitution. Then, by compatibility of substitutions with the typing judgement [31], we deduce* $\sigma\Gamma \vdash \sigma\mathit{adv} : \mathit{Advice}\ m\ a'\ b'$*. Therefore,* $(\mathit{unsafeCoerce}\ \mathit{adv})$ *corresponds exactly to* $\sigma\mathit{adv}$*, and is safe.* □

6 Open and Protected Modules, with Effects

This section illustrates how we can exploit the monadic embedding of aspects to encode Open Modules [2] extended with effects. Additionally, we present the notion of *protected pointcuts*, which are pointcuts whose type places restrictions on admissible advice. We illustrate the use of protected pointcuts to enforce control flow properties of external advice, reusing the approach of EffectiveAdvice [28].

6.1 A Simple Example

We first describe a simple example that serves as the starting point. Figure 2 describes a Fibonacci module, following the canonical example of Open Modules. The module uses an internal aspect to implement the recursive definition of Fibonacci: the base function, $\mathit{fibBase}$, simply implements the base case; and the fibAdv advice implements recursion when the pointcut pcFib matches. Note that pcFib uses the user-defined pointcut $\mathit{pcArgGT}$ (defined in Sect. 4.1) to check that the call to $\mathit{fibBase}$ is done with an argument greater than 2. The fib function is defined by first deploying the internal aspect, and then partially applying $\#$ to $\mathit{fibBase}$. This transparently ensures that an application

```
module MemoizedFib (fib) where
import qualified Fib
import AOP

memo proceed n =
  do table ← get
     if member n table
       then return (table ! n)
       else do y ← proceed n
               table′ ← get
               put (insert n y table′)
               return y

fib = do deploy (aspect Fib.pcFib memo)
         Fib.fib
```

Fig. 3. Memoized Fibonacci module

of *fib* is open. The *fib* function is exported, together with the *pcFib* pointcut, which can be used by an external module to advise applications of the internal *fibBase* function. Figure 3 presents a Haskell module that provides a more efficient implementation of *fib* by using a memoization advice. To benefit from memoization, a client only has to import *fib* from the *MemoizedFib* module instead of directly from the *Fib* module.

Note that if we consider that the aspect language only supports the *pcCall* pointcut designator, this implementation actually represents an open module proper. Preserving the properties of open modules, in particular protecting from external advising of internal functions, in presence of arbitrary quantification (*e.g.*, *pcType*, or an always-matching pointcut) is left for future work. Importantly, just like Open Modules, the approach described here does not ensure anything about the advice beyond type safety. In particular, it is possible to create an aspect that incorrectly calls *proceed* several times, or an aspect that has undesired computational effects. Fortunately, the type system can assist us in expressing and enforcing specific interference properties.

6.2 Protected Pointcuts

In order to extend Open Modules with effect-related enforcement, we introduce the notion of *protected pointcuts*, which are pointcuts enriched with restrictions on the effects that associated advice can exhibit. Simply put, a protected pointcut embeds a *combinator* that is applied to the advice in order to build an aspect. If the advice does not respect the (type) restrictions expressed by the combinator, the aspect creation expression simply does not typecheck and hence the aspect cannot be built. A combinator is any function that can produce an advice:

```
type Combinator t m a b = Monad m ⇒ t → Advice m a b
```

The *protectPC* function packs together a pointcut and a combinator:

$$protectPC :: (Monad\ m, LessGen\ (a \rightarrow m\ b)\ (c \rightarrow m\ d)) \Rightarrow \\ \quad PC\ m\ a\ b \rightarrow Combinator\ t\ m\ c\ d \rightarrow ProtectedPC\ m\ a\ b\ t\ c\ d$$

A protected pointcut, of type $ProtectedPC$, cannot be used with the standard aspect creation function $aspect$. The following $pAspect$ function is the only way to get an aspect from a protected pointcut (the constructor PPC is not exposed):

$$pAspect :: Monad\ m \Rightarrow ProtectedPC\ m\ a\ b\ t\ c\ d \rightarrow t \rightarrow Aspect\ m\ a\ b\ c\ d \\ pAspect\ (PPC\ pc\ comb)\ adv = aspect\ pc\ (comb\ adv)$$

The key point here is that when building an aspect using a protected pointcut, the combinator $comb$ is applied to the advice adv. We now show how to exploit this extension of Open Modules to restrict control flow properties, using the proper type combinators. The next section describes how to control computational effects.

6.3 Enforcing Control Flow Properties

Rinard *et al.* present a classification of advice in four categories depending on how they affect the control flow of programs [34]:

- **Combination:** The advice can call $proceed$ any number of times.
- **Replacement:** There are no calls to $proceed$ in the advice.
- **Augmentation:** The advice calls $proceed$ exactly once, and it does not modify the arguments to or the return value of $proceed$.
- **Narrowing:** The advice calls $proceed$ at most once, and does not modify the arguments to or the return value of $proceed$.

In EffectiveAdvice [28], Oliveira and colleagues show a type-based enforcement of these categories, through advice combinators (Fig. 4). These combinators fit the general $Combinator$ type we described in Sect. 6.2, and can therefore be embedded in protected pointcuts. Observe that no combinator is needed for combination advice, because no interference properties are enforced. Replacement advice is advice that has no access to $proceed$. Augmentation advice is represented by a pair of *before*/*after* advice functions, such that *after* has access to the argument, the return value, and an extra value optionally exposed by the *before* function. A narrowing advice is in fact the combination of both a replacement advice and an augmentation advice, where the choice between both is driven by a runtime predicate.

As an illustration, observe that memoization is a typical example of a narrowing advice: the combination of a replacement advice ("return memoized value without proceeding") and an augmentation advice ("proceed and memoize return value"), where the choice between both is driven by a runtime predicate ("is there a memoized value for this argument?"). Therefore, it is now straightforward for the Fib module to expose a protected pointcut that restricts valid advice to narrowing advice only:

$$\mathbf{module}\ Fib\ (fib, ppcFib)\ \mathbf{where} \\ ppcFib = protectPC\ pcFib\ narrow \\ \ldots$$

```
type Replace m a b = (a → m b)
replace :: Replace m a b → Advice m a b
replace radv proceed = radv

type Augment a b c m = (a → m c, a → b → c → m ())
augment :: Monad m ⇒ Augment a b c m → Advice m a b
augment (before, after) proceed arg =
  do c ← before arg
     b ← proceed arg
     after arg b c
     return b

type Narrow m a b c = (a → m Bool, Augment m a b c, Replace m a b)
narrow :: Monad m ⇒ Narrow m a b c → Advice m a b
narrow (p, aug, rep) proceed x =
  do b ← p x
     if b then replace rep proceed x
          else  augment aug proceed x
```

Fig. 4. Replacement, augmentation and narrowing advice combinators (adapted from [28])

```
memo :: (𝕊_M (Map a b) m, Ord a) ⇒ Narrow a b () m
memo = (pred, (before, after), rep) where
  pred n      = do {table ← get; return (member n table)}
  before _    = return ()
  after n r _ = do {table ← get; put (insert n r table)}
  rep x       = do {table ← get; return (table ! n)}
```

Fig. 5. Memoization as a narrowing advice (adapted from [28])

The protected pointcut *ppcFib* embeds the *narrow* type combinator. Hence, only advice that can be statically typed as narrowing advice can be bound to that pointcut. A valid definition of the *memo* advice is given in Fig. 5. Note that the protected pointcut is only restrictive with respect to the control flow effect of the advice, but not with respect to its computational effect: any monad m is accepted.

Finally, note that this approach is not limited to the four categories of Rinard *et al.*; custom kinds of advice can be defined in a similar way. For instance, we consider *adaptation* advice as a weaker version of narrowing where the advice is allowed to modify the arguments to *proceed*. The implementation is straightforward:

```
type Adaptation a b c m = (a → a, a → m c, a → b → c → m ())
adapt :: Adaptation a b c m → Advice m a b
adapt (adapter, before, after) proceed arg =
  augment (before, after) proceed (adapter arg)
```

A relevant design choice is whether the *adapter* function is pure or is allowed to perform effects. This choice affects which properties can be statically checked based on

the type of the advice. Allowing effects is more expressive, but it is source of potential interferences, in addition to advices and pointcuts. The next section describes how to control effect interference between these components.

7 Controlling Effect Interference

The monadic embedding of aspects also enables reasoning about computational effects. We are particularly interested in reasoning about *effect interference* between components of a system: aspects, base programs, and combinations thereof. To do this, in Section 7.1 we first show how to adapt the non-interference types defined in EffectiveAdvice [28], which distinguish between aspect and base computation. The essence of this technique is to use parametricity to forbid components from making assumptions about some part of the monad stack. Then, because components must work uniformly over the restricted section of the stack, they can only utilize effects available in the non-restricted section.

However, this approach falls short when considering several aspects in a system, because aspects (and base programs) can still interfere between them. In Section 7.2 we show how a refinement of the technique can be used to address this situation, but that unfortunately is impractical because it requires explicit liftings and strongly couples components to particular shapes of the monad stack—hampering modularity and reusability.

Finally, we show in Section 7.4 a different approach to enforce non-interference based on *monad views* [35], a recently developed mechanism for handling the monad stack, which is summarized in Section 7.3.

7.1 Distinguishing Aspect and Base Computation

To illustrate the usefulness of distinguishing between aspect and base computation, consider a Fibonacci module where the internal calls throw an exception when given a negative integer as argument. In that situation, it is interesting to ensure that the external advice bound to the exposed pointcut cannot throw or catch those exceptions.

Following EffectiveAdvice [28], we can enforce an advice to be parametric with respect to a monad used by base computation, effectively splitting the monad stack into two. To this end we define the $\mathbb{NIA}_\mathrm{T}$ ($\mathbb{NI}$ stands for non-interference) type:

$$\textbf{newtype}\ \mathbb{NIA}_\mathrm{T}\ t\ m\ a = \mathbb{NIA}_\mathrm{T}\ (\mathbb{S}_\mathrm{T}\ (\mathit{AspectEnv}\ (\mathbb{NIA}_\mathrm{T}\ t\ m))\ (t\ m)\ a)$$

Observe that $\mathbb{NIA}_\mathrm{T}$ splits the monad stack into an upper part t, with the effects available to aspects; and a lower part m, with the effects available to base computation. We extend other definitions (weave, deploy, etc.) accordingly.

Note that $\mathbb{NIA}_\mathrm{T}$ is a proper monad, but not a monad transformer. This is because the $\mathit{MonadTrans}$ class is designed for a type constructor t that is applied to some monad m, but $\mathbb{NIA}_\mathrm{T}$ takes two types as arguments. We could define the partial application $\mathbb{NIA}_\mathrm{T}\ t$ as a monad transformer, but this is inconvenient because explicit *lift* operations would skip the upper layer of the stack[14]. However, for allowing explicit lifting into

[14] Because we would lift from m to $(\mathbb{NIA}_\mathrm{T}\ t)\ m$.

$\mathbb{NIA}_\mathrm{T}$ we need an operation to transform a computation from $t\ m$ into an $\mathbb{NIA}_\mathrm{T}\ t\ m$ computation. To this end we provide the *niLift* operation as follows:

$$
\begin{aligned}
&\mathit{niLift} :: \mathit{Monad}\ (t\ m) \Rightarrow t\ m\ a \rightarrow \mathbb{NIA}_\mathrm{T}\ t\ m\ a\\
&\mathit{niLift}\ ma = \mathbb{NIA}_\mathrm{T}\ \$\ \mathbb{S}_\mathrm{T}\ \$\ \lambda aenv \rightarrow \mathbf{do}\\
&\qquad a \leftarrow ma\\
&\qquad \mathit{return}\ (a, aenv)
\end{aligned}
$$

Effect Interference and Pointcuts. The novelty compared to EffectiveAdvice is that we also have to deal with interferences for pointcuts. But to allow effect-based reasoning on pointcuts, we need to distinguish between the monad used by the base computation and the monad used by pointcuts. Indeed, in the interpretation of the type $PC\ m\ a\ b$, m stands for both monads, which forbids to reason separately about them. To address this issue, we need to interpret $PC\ m\ a\ b$ differently, by saying that the matched type is $a \rightarrow b$ instead of $a \rightarrow m\ b$. In this way, the monad for the base computation (which is implicitly bound by b) does not have to be m at the time the pointcut is defined. To accommodate this new interpretation with the rest of the code, very little changes have to be made[15]. Mainly, the types of *pcCall*, *pcType* and the definition of *Aspect*:

$$
\begin{aligned}
&\mathit{pcCall}, \mathit{pcType} :: \mathit{Monad}\ m \Rightarrow (a \rightarrow b) \rightarrow PC\ m\ a\ b\\
&\mathbf{data}\ \mathit{Aspect}\ m\ a\ b\ c\ d = (\mathit{Monad}\ m, \mathit{LessGen}\ (a \rightarrow b)\ (c \rightarrow m\ d)) \Rightarrow\\
&\qquad \mathit{Aspect}\ (PC\ m\ a\ b)\ (\mathit{Advice}\ m\ c\ d)
\end{aligned}
$$

Note how the definition of *Aspect* forces the monad of the pointcut computation to be unified with that of the advice, and with that of the base code. The results of Sect. 5 can straightforwardly be rephrased with these new definitions.

Typing Non-interfering Pointcuts and Advices. Using rank-2 types [30] we can restrict the type of pointcuts and advices. The following types synonyms guarantee that non-interfering pointcuts (*NIPC*) and advices (*NIAdvice*) only use effects available in t.

$$
\begin{aligned}
&\mathbf{type}\ \mathit{NIPC}\ t\ a\ b = \forall m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t) \Rightarrow\\
&\qquad PC\ (\mathbb{NIA}_\mathrm{T}\ t\ m)\ a\ b\\
&\mathbf{type}\ \mathit{NIAdvice}\ t\ a\ b = \forall m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t) \Rightarrow\\
&\qquad \mathit{Advice}\ (\mathbb{NIA}_\mathrm{T}\ t\ m)\ a\ b
\end{aligned}
$$

By universally quantifying over the type m of the effects used in the base computation, these types enforce, through the properties of parametricity, that pointcuts or advices cannot refer to specific effects in the base program. We can define aspect construction functions that enforce different (non-)interference patterns, such as non-interfering pointcut *NIPC* with unrestricted advice *Advice*, unrestricted pointcut *PC* with non-interfering advice *NIAdvice*, etc.

[15] The implementation available online uses this interpretation of $PC\ m\ a\ b$.

$$
\begin{array}{l}
\textbf{module}\ \mathit{FibErr}\ (\mathit{fib}, \mathit{ppcFib})\ \textbf{where} \\
\textbf{import}\ \mathit{AOP} \\
\mathit{pcFib} = \mathit{pcCall}\ \mathit{fibBase}\ \text{`}\mathit{pcAnd}\text{`}\ \mathit{pcArgGT}\ 2 \\
\mathit{ppcFib} = \mathit{protectPC}\ \mathit{pcFib}\ \mathit{niAdvice} \\
\mathit{fibBase}\ n = \mathit{return}\ 1 \\
\mathit{fibAdv}\ \mathit{proceed}\ n = \textbf{do}\ \mathit{f1} \leftarrow \mathit{errorFib}\ \#\ (n-1) \\
\qquad\qquad\qquad\qquad\quad \mathit{f2} \leftarrow \mathit{errorFib}\ \#\ (n-2) \\
\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{f1} + \mathit{f2}) \\
\mathit{fib} = \textbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ \mathit{fibAdv}) \\
\qquad\qquad \mathit{return}\ \mathit{errorFib} \\
\mathit{errorFib} :: (\mathit{MonadTrans}\ t, \mathbb{E}_{\mathrm{M}}\ \mathit{String}\ m) \Rightarrow \mathit{Int} \rightarrow \mathbb{NIA}_{\mathrm{T}}\ t\ m\ \mathit{Int} \\
\mathit{errorFib}\ n = \textbf{if}\ n < 0 \\
\qquad\qquad\qquad \textbf{then}\ \ (\mathit{niLift} \circ \mathit{lift} \circ \mathit{throwError})\,\text{“}\mathit{Error} : \mathit{negative\ argument}'' \\
\qquad\qquad\qquad \textbf{else}\ \ \ \mathit{fibBase}\ \#\ n
\end{array}
$$

Fig. 6. Fibonacci with error

Enforcing Non-interference. Coming back to Open Modules and protected pointcuts, to enforce non-interfering advice we need to define a typed combinator that requires an advice of type $\mathit{NIAdvice}$:

$$
\begin{array}{l}
\mathit{niAdvice} :: (\mathit{Monad}\ (t\ m), \mathit{Monad}\ m) \Rightarrow \\
\qquad\qquad \mathit{NIAdvice}\ t\ a\ b \rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}}\ t\ m)\ a\ b \\
\mathit{niAdvice}\ \mathit{adv} = \mathit{adv}
\end{array}
$$

Observe that the $\mathit{niAdvice}$ combinator is computationally the identity function, but it does impose a type requirement on its argument. Using this combinator, a module can expose a protected pointcut that enforces non-interference with base effects.

Fibonacci Module with Error Handling. We now define a Fibonacci module (Fig. 6) where base functions $\mathit{fibBase}$ and fibAdv raise an exception when given a negative argument.[16] The exception is raised on monad m that corresponds to base computation, and which is required to be an instance of $\mathbb{E}_{\mathrm{M}}$. The definition of ppcFib enforces that external advice cannot manipulate exceptions in m, because it uses the $\mathit{niAdvice}$ advice combinator. The drawback is that because we are using an effect in an inner layer of the stack, we need to use explicit lifting to satisfy the expected type.

Non-interfering Base Computation. Symmetrically, we can check that a part of the base code cannot interfere with effects available to aspects by using the type synonym NIBase, which universally quantifies over the type t of effects available to the advice:

$$
\begin{array}{l}
\textbf{type}\ \mathit{NIBase}\ m\ a\ b = \forall t.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t, \mathit{Monad}\ (t\ m)) \Rightarrow \\
\qquad\qquad\qquad\qquad a \rightarrow \mathbb{NIA}_{\mathrm{T}}\ t\ m\ b
\end{array}
$$

[16] We do not use an error-checking argument on purpose, for the sake of illustration. We use such an aspect in Sect. 7.2 where we consider the issues of multiple effectful aspects.

Reasoning about Pointcut Interference. Another use of effect reasoning can be done at the level of pointcuts. Indeed, in the monadic embedding of aspects, we allow for effectful pointcuts. For example, we can define a sequential pointcut combinator [10] $pcSeq\ pc_1\ pc_2$, that matches first pc_1 and then pc_2:

$$
\begin{array}{l}
pcSeq :: (\mathbb{S}_{\mathrm{M}}\ Bool\ m) \Rightarrow PC\ m\ a\ b \rightarrow PC\ m\ c\ d \rightarrow PC\ m\ c\ d \\
pcSeq\ (PC\ mpc_1)\ (PC\ mpc_2) = \\
\quad PC\ \$\ \mathbf{do}\ pc_1 \leftarrow mpc_1 \\
\qquad\qquad pc_2 \leftarrow mpc_2 \\
\qquad\qquad return\ \$\ \lambda jp \rightarrow \mathbf{do}\ b \leftarrow get \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{if}\ b\ \mathbf{then}\ pc_2\ jp \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \ \mathbf{do}\ b' \leftarrow pc_1\ jp \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad put\ b' \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad return\ False
\end{array}
$$

As expressed in the $\mathbb{S}_{\mathrm{M}}\ \ Bool\ \ m$ constraint, the pointcut requires a boolean state in which to store the current point of its matching behavior: $False$ (resp. $True$) means pc_1 (resp. pc_2) is to be matched. Consequently, any base program that modifies this state will alter the behavior of the pointcut. This situation can be avoided by using the non-interfering base computation type $NIBase$, just described above.

7.2 Interference between Multiple Aspects

$\mathbb{NIA}_{\mathrm{T}}$ only distinguishes between base and aspect computation. Although useful, this implies that interference between aspects is still possible because all of them will share the same upper part of the monad stack. A similar situation happens with base programs and the lower part of the monad stack.

To illustrate this issue, consider a Fibonacci module program that uses the $memo$ advice to improve the performance, and also uses a $checkArg$ advice that throws an exception when given a negative argument (instead of a base code check as in Fig. 6). In this setting, $checkArg$ could update the cache with incorrect values, either accidentally or intentionally; or conversely, $memo$ could throw arbitrary exceptions, even with a non-negative argument.

Finer-Grained Splitting of the Monad Stack. Following the idea used in $\mathbb{NIA}_{\mathrm{T}}$, to enforce non-interference between $memo$ and $checkArg$ we need to split the monad stack into the monad for base computation m, and two upper layers t_1 and t_2. The idea is to assign to each aspect a unique layer in the stack, and to use parametricity to ensure non-interference. To this end we define the $\mathbb{NIA}_{\mathrm{T}_2}$ monad, which splits the monad stack as described. We also consider $niLift_2$, which serves the same role as $niLift$.

$$
\begin{array}{l}
\mathbf{newtype}\ \mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m\ a = \\
\qquad\qquad \mathbb{NIA}_{\mathrm{T}_2}\ (\mathbb{S}_{\mathrm{T}}\ (AspectEnv\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m))\ (t_1\ (t_2\ m))\ a)
\end{array}
$$

Again, we extend other definitions properly ($weave$, etc.). Using rank-2 types, the following type synonyms guarantee that non-interfering pointcuts and advices access can

only access the effect available in the first layer L_1, which corresponds to t_1; or in the second layer L_2, which corresponds to t_2.

$$
\begin{aligned}
&\mathbf{type}\ \mathit{NIPC}_{L_1}\ \ t_1\ a\ b = \forall t_2\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{PC}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\mathbf{type}\ \mathit{NIPC}_{L_2}\ \ t_2\ a\ b = \forall t_1\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{PC}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\mathbf{type}\ \mathit{NIAdvice}_{L_1}\ t_1\ a\ b = \forall t_2\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\mathbf{type}\ \mathit{NIAdvice}_{L_2}\ t_2\ a\ b = \forall t_1\ m.(\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1,\\
&\qquad\qquad \mathit{MonadTrans}\ t_2) \Rightarrow \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b
\end{aligned}
$$

Non-interference Combinators. To enforce non-interference properties we need to define advice combinators, as we did with $niAdvice$. Again, we can enforce different non-interference patterns, by defining as many construction functions as required. We describe the advice combinators $niAdvice_{L_1}$ and $niAdvice_{L_2}$ that enforce that aspects work exclusively with the effect provided by the first and second layer, respectively.

$$
\begin{aligned}
&\mathit{niAdvice}_{L_1} :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1, \mathit{MonadTrans}\ t_2) \Rightarrow\\
&\qquad \mathit{NIAdvice}_{L_1}\ t_1\ a\ b \to \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\mathit{niAdvice}_{L_1}\ \mathit{adv} = \mathit{adv}\\
&\mathit{niAdvice}_{L_2} :: (\mathit{Monad}\ m, \mathit{MonadTrans}\ t_1, \mathit{MonadTrans}\ t_2) \Rightarrow\\
&\qquad \mathit{NIAdvice}_{L_2}\ t_2\ a\ b \to \mathit{Advice}\ (\mathbb{NIA}_{\mathrm{T}_2}\ t_1\ t_2\ m)\ a\ b\\
&\mathit{niAdvice}_{L_2}\ \mathit{adv} = \mathit{adv}
\end{aligned}
$$

Now we define the monad stack S that provides the state and error-handling effects.

$$\mathbf{type}\ S = \mathbb{NIA}_{\mathrm{T}_2}\ (\mathbb{E}_{\mathrm{T}}\ \mathit{String})\ (\mathbb{S}_{\mathrm{T}}\ (\mathit{Map\ Int\ Int}))\ \mathbb{I}$$

Then, we define the new fibonacci function using the $checkArg_{L_1}$ and $memo_{L_2}$ advices, which operate on the first and second layer of the monad stack, respectively.

$$
\begin{aligned}
&\mathit{fibMemoErr} :: \mathit{Int} \to S\ \mathit{Int}\\
&\mathit{fibMemoErr}\ n = \mathbf{do}\ \mathit{deploy}\ (\mathit{aspect}\ \mathit{pcFib}\ (\mathit{niAdvice}_{L_2}\ \mathit{memo}_{L_2}))\\
&\qquad f \leftarrow \mathit{fib}\\
&\qquad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ f)\ (\mathit{niAdvice}_{L_1}\ \mathit{checkArg}_{L_1}))\\
&\qquad f\ \#\ n
\end{aligned}
$$

The implementation of $checkArg_{L_1}$ is as follows:

$$
\begin{aligned}
&\mathit{checkArg}_{L_1}\ \mathit{proceed}\ \mathit{arg} =\\
&\quad \mathbf{if}\ \mathit{arg} < 0\\
&\qquad \mathbf{then}\ (\mathit{niLift}_2 \circ \mathit{throwError})\ \texttt{"Error: negative argument"}\\
&\qquad \mathbf{else}\ \mathit{proceed}\ \mathit{arg}
\end{aligned}
$$

And similarly, we define $memo_{L_2}$:

$$
\begin{array}{l}
memo_{L_2}\ proceed\ n = \\
\quad \mathbf{do}\ table \leftarrow niLift_2\ \$\ lift\ \$\ get \\
\quad\quad \mathbf{if}\ member\ n\ table \\
\quad\quad\quad \mathbf{then}\ return\ (table\ !\ n) \\
\quad\quad\quad \mathbf{else}\ \ \mathbf{do}\ y \leftarrow proceed\ n \\
\quad\quad\quad\quad\quad\quad table' \leftarrow niLift_2\ \$\ lift\ \$\ get \\
\quad\quad\quad\quad\quad\quad (niLift_2 \circ lift \circ put)\ (insert\ n\ y\ table') \\
\quad\quad\quad\quad\quad\quad return\ y
\end{array}
$$

Note that $checkArg_{L_1}$ is applied on calls to the *external* fibonacci function f, while $memo_{L_2}$ is applied to the *internal* calls of the Fibonacci module, exposed by $pcFib$.

While an improvement over the binary base/aspect approach of EffectiveAdvice, illustrated in Section 7.1, this approach has two major drawbacks. First, it is not scalable because we need a different $\mathbb{NIA}_{\mathrm{T}_n}$ monad to support a setting with n mutually exclusive effects for aspects. Second, it is necessary to use explicit lifting in the implementation of advice. The reason is that we are explicitly using an effect from a layer at an arbitrary position in monad stack. Because we need to preserve parametricity to enforce non-interference, an advice cannot make *any* assumptions on the monad transformers that compose the stack. In particular, it cannot assume that the transformers support implicit liftings from the inner layers of the stack. In fact, in the presence of implicit lifting the layer from which an effect comes depends on the concrete monad stack used. These issues hamper modularity and reusability of aspects. In general, there is a tension between implicit lifting—designed to make a layer provide several effects at once—and splitting the monad stack with one aspect/effect per layer. In Section 7.4 we address these issues by using monad views [35].

7.3 Interlude: Monad Views

Monad views, recently developed by Schrijvers and Oliveira [35], are a technique for handling the monad stack, which extends and complements the standard mechanisms of explicit and implicit liftings (Section 2). Monad views provide robust support for accessing the effects of the monad stack without being coupled to a particular stack layout. Views are denoted using $\rightsquigarrow$, and are an instance of the $View$ type class that defines the *from* operation. Additionally, we use *bidirectional views*, denoted with the $\bowtie$ type operator. In addition to *from*, a bidirectional view supports the to operation.

$$
\begin{array}{lll}
from :: (Monad\ m, Monad\ n, View\ (\rightsquigarrow)) & \Rightarrow n \rightsquigarrow m \rightarrow n\ a \rightarrow m\ a \\
to\ \ \ \ :: (Monad\ m, Monad\ n) & \Rightarrow n \bowtie m \rightarrow m\ a \rightarrow n\ a
\end{array}
$$

In short, given two monads n and m, a view $n \rightsquigarrow m$ transforms computations from n to m, and a bidirectional view $n \bowtie m$ can also transform computations from m to n.

View-Specific Operations. Views are first-class values, hence they can be used as arguments. For instance, consider the functions $getv$ and $putv$ defined in [35]:

$$
\begin{array}{l}
getv :: (Monad\ m, \mathbb{S}_{\mathrm{M}}\ s\ n, View\ (\rightsquigarrow)) \Rightarrow (n \rightsquigarrow m) \rightarrow m\ s \\
getv\ v = from\ v\ \$\ get
\end{array}
$$

$$putv :: (Monad\ m, \mathbb{S}_M\ s\ n, View\ (\rightsquigarrow)) \Rightarrow (n \rightsquigarrow m) \rightarrow s \rightarrow m\ ()$$
$$putv\ v = from\ v \circ put$$

Given an initial monad m and a view $n \rightsquigarrow m$, $getv$ returns a computation $m\ s$ from an arbitrary state layer n. Conversely, $putv$ puts a new value into state layer n.

Creating Views. Schrijvers and Oliveira propose the construction of views using *structural* and *nominal masks*, which are applied onto the layers of a monad stack [35].

- A structural mask is a bit-like mask applied to the monadic stack in order to hide the layers that conflict with implicit lifting. Such a mask is created by concatenating unary masks for each layer using the ::: type operator:[17] □ indicates a visible layer and ■ a hidden layer.
- A nominal mask refers to layers of the stack using *names* instead of relative positions. This is done with the *tag monad transformer* $\mathbb{T}$. Given an arbitrary type Tag, the layer $\mathbb{T}^{Tag}$ labels a particular position of the monad stack using type Tag. An example of a tagged monad stack (for some types Tag_1 and Tag_2) is:

  $$\textbf{type}\ M = \mathbb{T}^{Tag_1}\ (\mathbb{S}_T\ Int\ (\mathbb{T}^{Tag_2}\ \mathbb{E}_T\ String\ \mathbb{I}))$$

  where the $\mathbb{S}_T$ layer is labeled with Tag_1 and the $\mathbb{E}_T$ layer is labeled with Tag_2. For inspecting tagged monad stacks, the type class $n \sqsubseteq_{Tag} m$ exposes a monad n representing the layer of the stack m tagged with type Tag. It also provides the $structure$ operation to obtain the view between n and m associated to t:

  $$\textbf{class}\ (Monad\ m, Monad\ n) \Rightarrow n \sqsubseteq_{Tag} m\ \textbf{where}$$
  $$structure :: View\ (\rightsquigarrow) \Rightarrow Tag \rightarrow (n \rightsquigarrow m)$$

7.4 Beyond the Aspect/Base Distinction

Monad views enable a different approach to enforce non-interference. The idea is that aspects will be generic with respect to the effects they require using type class constraints, assuming exclusive access to a monad stack with those effects. To avoid non-interference, client code uses a concrete monad stack and transforms each advice into a view-specific advice where the aspect only sees the sections of the monad stack that it is allowed to access.

For instance, the $memo$ advice described in Fig. 3 requires access to a dictionary to store the precomputed results. This is explicit in the (inferred) type of the advice:

$$memo :: (Monad\ m, Ord\ a, \mathbb{S}_M\ (Map\ a\ b)\ m) \Rightarrow Advice\ m\ a\ b$$

In a similar way we define $checkArg$, which requires access to an error effect:

```
checkArg :: (Monad m, Num a, E_M String m) ⇒ Advice m a b
checkArg proceed arg =
  if arg < 0
    then throwError "Error : negative argument"
    else proceed arg
```

[17] We follow the graphical notation used in [35].

Fig. 7. Applying structural masks to the monad stack S_1

Arbitrarily Splitting the Monad Stack with Views. Observe now that the advice does not depend on the specific position of an effect in the monad stack. The novelty with respect to using implicit liftings is that we can assign to each aspect a *virtual view* of the monad stack that only contains the effect available to them. To assign a part of the monad stack to an advice we define the *withView* function:

$$withView :: (Monad\ n, Monad\ m) \Rightarrow n \bowtie m \to Advice\ n\ a\ b \to Advice\ m\ a\ b$$
$$withView\ v\ adv\ proceed\ arg = from\ v\ \$\ adv\ (\lambda a \to to\ v\ (proceed\ a))\ arg$$

This function transforms an advice from a restricted monad n to an advice in the "complete" stack m, using a bidirectional view provided as argument. We require a bidirectional view because we need to lift the *proceed* function, with type $a \to n\ b$ into an equivalent function with type $a \to m\ b$—which by construction performs effects only on n. Then, because evaluation of the restricted advice yields a computation $n\ b$, we use the *from* operation to lift it into a computation $m\ b$.

Observe that partially applying *withView* with a given view yields a function of type $Advice\ n\ a\ b \to Advice\ m\ a\ b$, which fits with the notion of advice combinators (Sect. 6.2). Therefore, it is possible to export protected pointcuts that expose a particular section of the monad stack to external advice. Additionally, we can define functions to transform join points and pointcuts, in a similar way to *withView*.

Using Structural Masks. Consider a concrete monad stack S_1 which holds the required state and error effects.

$$\textbf{type}\ S_1 = \mathbb{A}_{\mathrm{T}}\ (\mathbb{E}_{\mathrm{T}}\ String\ (\mathbb{S}_{\mathrm{T}}\ (Map\ Int\ Int)\ \mathbb{I}))$$

Then, we define the fibonacci function as follows:

$$\begin{aligned} fibMemoErr'\ n = \textbf{do}\ & deploy\ (aspect\ pcFib\ (withView\ v_1\ memo)) \\ & f \leftarrow fib \\ & deploy\ (aspect\ (pcCall\ f)\ (withView\ v_2\ checkArg)) \\ & f \mathbin{\#} n \\ \textbf{where}\ v_1 &= \square ::: \blacksquare ::: \square \\ v_2 &= \square ::: \square ::: \blacksquare \end{aligned}$$

We define views v_1 and v_2 using structural masks. Both allow access to $\mathbb{A}_\mathrm{T}$, allowing AOP-specific operations into advice (*e.g.*, deploying aspects). Besides that, v_1 exposes only the $\mathbb{S}_\mathrm{T}$ transformer, whereas v_2 only allows accessing to the $\mathbb{E}_\mathrm{T}$ transformer. Figure 7 depicts how views v_1 and v_2 define new *virtual monad stacks*, by applying structural masks to S_1. Note that structural masks can be applied only to monad transformers, but not the monad at the bottom of the stack.

It is clear that now aspects do not need to perform explicit liftings and are not coupled to a particular monad stack. However, these issues are present when constructing views using nominal masks. Changes to the monad stack that is used to run client code need to be reflected in (potentially many) client functions that use structural masks.

Using Nominal Masks. A more flexible approach that is not coupled to any particular monad stack is to use nominal masks to tag each effect required by aspects. Then client code can use the tags to directly access the effects and properly transform the advices. Consider a monad stack S_2, where the state and error layers are tagged:

$$
\begin{aligned}
&\mathbf{data}\ \mathit{StateTag}\\
&\mathbf{data}\ \mathit{ErrorTag}\\
&\mathbf{type}\ S_2 = \mathbb{A}_\mathrm{T}\ (\mathbb{T}^{\mathit{ErrorTag}}\ (\mathbb{E}_\mathrm{T}\ \mathit{String}\ (\mathbb{T}^{\mathit{StateTag}}\ (\mathbb{S}_\mathrm{T}\ (\mathit{Map\ Int\ Int})\ \mathbb{I})))
\end{aligned}
$$

The stack is tagged at the type level, therefore we define two singleton types (with no data constructors), namely $\mathit{StateTag}$ and $\mathit{ErrorTag}$, to use as arguments for the $\mathbb{T}$ monad transformer.

The fibonacci function implemented using nominal masks is:

$$
\begin{aligned}
&\mathit{fibMemoErr}'' :: \forall m\ n_1\ n_2.(\mathit{Monad}\ m,\\
&\quad n_1 \sqsubseteq_{\mathit{StateTag}} (\mathbb{A}_\mathrm{T}\ m), \mathbb{S}_\mathrm{M}\ (\mathit{Map\ Int\ Int})\ n_1,\\
&\quad n_2 \sqsubseteq_{\mathit{ErrorTag}} (\mathbb{A}_\mathrm{T}\ m), \mathbb{E}_\mathrm{M}\ \mathit{String}\ n_2)\\
&\quad \Rightarrow \mathit{Int} \rightarrow \mathbb{A}_\mathrm{T}\ m\ \mathit{Int}\\
&\mathit{fibMemoErr}''\ n = \mathbf{do}\ \mathit{deploy}\ (\mathit{aspect\ pcFib}\ (\mathit{withView}\ v_1\ \mathit{memo}))\\
&\qquad\qquad\qquad\qquad f \leftarrow \mathit{fib}\\
&\qquad\qquad\qquad\qquad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ f)\ (\mathit{withView}\ v_2\ \mathit{checkArg}))\\
&\qquad\qquad\qquad\qquad f \mathbin{\#} n\\
&\quad \mathbf{where}\ v_1 = \mathit{structure\ StateTag} :: n_1 \bowtie m\\
&\qquad\qquad\ \ v_2 = \mathit{structure\ ErrorTag} :: n_2 \bowtie m
\end{aligned}
$$

In contrast to the previous definition, we need to use explicit type annotations because using nominal masks can lead to ambiguity in type inference[18]. Observe that we assume a monad m that is tagged with two singleton types $\mathit{StateTag}$ and $\mathit{ErrorTag}$. We use $\sqsubseteq$ to expose these layers as monads n_1 and n_2 respectively, and we constrain these monads to expose the corresponding effects. Therefore, by using nominal masks we can independently evolve the definition of S_2, as long as we keep the tagged layers expected by $\mathit{fibMemoErr}''$ (satisfying both the tag name and the required effect).

[18] The $\forall m\ n_1\ n_2$ annotation is required to use the type variables in the scope of a $\mathbf{do}$ expression.

Perspectives on Using Views. The content of the **do** expression is the same using structural or nominal masks. In fact it is possible to define a more generic function that takes views v_1 and v_2 as argument. Because views are first-class values, there is a wide design space on how to use them to control aspect interference. For example, aspects can be defined directly using $\sqsubseteq$ constraints as required.

On the other hand, programmers must carefully define the views that are provided to each advice, because the typechecker cannot distinguish between intentional and accidental sharing of effects.

Controlling effect interference between aspects is a well-known and widely researched area in the AOP community. The two approaches presented in this section show that the concrete mechanism used to manage the monad stack determines the expressiveness of type-based reasoning techniques. We believe that the problem of assigning exclusive access to effects in the monadic stack originates from the fact that the monad stack is *public* and *transparent* to all components in a system. We conjecture that a mechanism that statically controls access to effects, while being flexible for developers ought to be devised, and indeed is a line of future work that transcends aspect-oriented programming. An additional line of future work is the connection between monad views and MRI [29] (a framework for monadic reasoning that extends EffectiveAdvice [35]), which is based on parametricity and considers only implicit and explicit lifting.

As a final remark, in a setting with an unrestricted $deploy$ operation the restrictions on advice must be applied at each particular aspect deployment. This makes it difficult to establish global properties about advice in a system (which may require external static analysis). This can be solved with a custom $\mathbb{A}_{\mathrm{T}}$-like monad transformer that provides a more restricted deployment mechanism.

8 Language Extensions

The typed monadic embedding of aspects supports modular extensions of the aspect language. The simplest extension is to introduce new user-defined pointcuts. More interestingly, because the language features a monadic weaver [38], we can modularly implement new semantics for aspect scoping and weaving. In addition, all language extensions benefit from the type-based reasoning techniques described in this paper—to the best of our knowledge, this is a novel contribution of this work. In this section we describe the following developments:

- A user-defined $pcCflow$ pointcut designator.
- Secure weaving, in which a set of join points can be hidden from advising.
- Privileged aspects that can see hidden join points from a secure computation.
- Aspect weaving with *execution levels* [41].
- An example of type-based reasoning in the semantics of execution levels.

8.1 Cflow Pointcut

An interesting illustration of extending the language with user-defined pointcuts is the case of control flow checks. Essentially, implementing the $pcCflow$ pointcut requires a

way to track join points emitted during program execution. This tracking mechanism can be implemented modularly using a state monad transformer that holds a stack of join points, and an aspect that matches every join point, stores it in the stack, and then proceeds to obtain the result, which is returned after popping the stack. This corresponds to the stack-based implementation of *cflow* described in [24].

Join Point Stack. To do this, we first define a join point stack as a list of existentially-quantified join points, EJP, just like we did to define the aspect environment as a list of homogeneous $EAspect$ values (Sect. 3.1).

$$\textbf{data}\ EJP = \forall a\ b\ m.Monad\ m \Rightarrow EJP\ (JP\ m\ a\ b)$$
$$\textbf{type}\ JPStack = [EJP]$$

Then, to collect the join points into a $JPStack$ we define the $\mathbb{JP}_{\mathrm{T}}$ monad transformer, reusing the implementation of the standard $\mathbb{S}_{\mathrm{T}}$ transformer:

$$\textbf{newtype}\ \mathbb{JP}_{\mathrm{T}}\ m\ a = \mathbb{JP}_{\mathrm{T}}\ (\mathbb{S}_{\mathrm{T}}\ JPStack\ m\ a)$$

In addition, to support a polymorphic monad stack we define the $\mathbb{JP}_{\mathrm{M}}$ type class as follows, and declare $\mathbb{JP}_{\mathrm{T}}$ as an instance.

$$\begin{array}{l}
\textbf{class}\ Monad\ m \Rightarrow \mathbb{JP}_{\mathrm{M}}\ m\ \textbf{where} \\
\quad getJPStack\ \ :: m\ JPStack \\
\quad pushJPStack :: EJP \rightarrow m\ () \\
\quad popJPStack\ \ :: m\ () \\
\textbf{instance}\ Monad\ m \Rightarrow \mathbb{JP}_{\mathrm{M}}\ (\mathbb{JP}_{\mathrm{T}}\ m)\ \textbf{where} \ldots
\end{array}$$

Defining pcCflow. Given the definitions above, the implementation of $pcCflow$ is very similar to that of $pcCall$ (Sect. 3.1).[19]

$$\begin{array}{l}
pcCflow :: \mathbb{JP}_{\mathrm{M}}\ m \Rightarrow (a \rightarrow m\ b) \rightarrow PC\ m\ c\ (m'\ d) \\
pcCflow\ f = return\ (\lambda_ \rightarrow \textbf{do} \\
\quad jpStack \leftarrow getJPStack \\
\quad return\ \$\ any\ (\lambda ejp \rightarrow compareFunEJP\ f\ ejp \wedge compareTypeEJP\ f\ ejp) \\
\qquad\qquad\qquad jpStack
\end{array}$$

Here $compareFunEJP$ checks the equality of the function bound to the join point and function f; and $compareTypeEJP$ checks that the type of f is more general than the type of the join point. Function any returns whether any element of jps satisfies a given predicate. We can define the $pcCflowbelow$ pointcut in a similar way.

Maintaining the Join Point Stack. Now it remains to define the aspect that maintains the join point stack. We first define the $pcAny$ pointcut, which matches all functions applications and pushes the corresponding join point into the stack.

[19] Note that, as discussed in Sect. 4.1, we specifically declare that the matched type of the pointcut is in a different monad m'.

$$
\begin{aligned}
&pcAny :: \mathbb{JP}_{\mathrm{M}}\ m \Rightarrow PC\ m\ a\ b\\
&pcAny = PC\ \$\ return\ \$\ \lambda jp \rightarrow \mathbf{do}\ pushJPStack\ (EJP\ jp)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ True
\end{aligned}
$$

Note that the definition of *pcAny* preserves type soundness (Sect. 4.1) because its matched type is given by two fresh type variables a and b, and hence is the most general type possible. Next, we define *collectAdv* as an advice that performs *proceed*, pops the stack and returns the result.

$$
\begin{aligned}
&collectAdv\ proceed\ arg = \mathbf{do}\ result \leftarrow proceed\ arg\\
&\qquad\qquad\qquad\qquad\qquad\quad popJPStack\\
&\qquad\qquad\qquad\qquad\qquad\quad return\ result
\end{aligned}
$$

Finally, we define the *maintainJpStack* aspect as follows.

$$
\begin{aligned}
&maintainJpStack :: \mathbb{JP}_{\mathrm{M}}\ m \Rightarrow Aspect\ m\ a\ (m\ b)\ a\ b\\
&maintainJpStack = aspect\ pcAny\ collectAdv
\end{aligned}
$$

This approach is inefficient because we are matching and storing all join points, instead of only those that can be queried in existing uses of *pcCflow*. Alternative optimizations can be defined, for example putting in the stack only relevant join points, or a per-flow deployment that allows using a boolean instead of a stack [24].

A consequence of not defining *pcCflow* as a *primitive* pointcut is that we need to ensure that evaluation of *maintainJpStack* occurs before than any other advice. Otherwise, control flow pointcuts from other aspects will have incorrect information to determine whether to execute the advice. This can be implemented directly in a custom $\mathbb{A}_{\mathrm{T}}$ transformer that takes a list of priority aspects and ensures they are always evaluated first during weaving.

8.2 Secure Weaving

For security reasons it can be interesting to protect certain join points from being advised. To support such a secure weaving, we define a new monad transformer $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$, which embeds an (existentially quantified) pointcut that specifies the hidden join points, and we modify the weaving process accordingly (not shown here).

$$
\begin{aligned}
&\mathbf{data}\ EPC\ m = \forall a\ b.EPC\ (PC\ m\ a\ b)\\
&\mathbf{data}\ \mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m\ a = \mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ (AspectEnv\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m)\\
&\qquad\qquad\qquad \rightarrow EPC\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m)\\
&\qquad\qquad\qquad \rightarrow m\ (a, (AspectEnv\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m), EPC\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m))))
\end{aligned}
$$

This can be particularly useful when used with the *pcCflow* pointcut to protect the computation that occurs in the control flow of critical function applications. For instance, we can ensure that the whole control flow of function f is protected from advising during the execution of program p , assuming a function $run\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$, similar to $run\mathbb{A}_{\mathrm{T}}$ (Sect. 3.2):

$$
run\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ (EPC\ (pcCflow\ f))\ p
$$

8.3 Privileged Aspects

Hiding some join points to *all* aspects may be too restrictive. For instance, certain "system" aspects like access control should be treated as privileged and view all join points. Another example is the aspect in charge of maintaining the join point stack for the sake of control flow reasoning (used by $pcCflow$). In such cases, it is important to be able to define a set of privileged aspects, which can advise all join points, even those that are normally hidden in a secure computation. The implementation of a privileged aspects list is a straightforward extension to the secure weaving mechanism described above.

8.4 Execution Levels

Execution levels avoid unwanted *computational interference* between aspects, *i.e.* when an aspect execution produces join points that are visible to others, including itself [41]. Execution levels give structure to execution by establishing a tower in which the flow of control navigates. Aspects are deployed at a given level and can only affect the execution of the underlying level. The execution of an aspect (both pointcuts and advices) is therefore not visible to itself and to other aspects deployed at the same level, only to aspects standing one level above. The original computation triggered by the last $proceed$ in the advice chain is always executed at the level at which the join point was emitted. If needed, the programmer can use level-shifting operators to move execution up and down in the tower.

The monadic semantics of execution levels are implemented in the $\mathbb{EL}_\mathrm{T}$ monad transformer (Fig. 8). The $Level$ type synonym represents the level of execution as an integer. $\mathbb{EL}_\mathrm{T}$ wraps a run function that takes an initial level and returns a computation in the underlying monad m, with a value of type a and a potentially modified level. As in the $\mathbb{A}_\mathrm{T}$ transformer, the monadic $bind$ and $return$ functions are the same as in the state monad transformer. The private operations inc, dec, and at are used to define the user-visible operations $current$, up, $down$, and $lambda_at$. In addition to level shifting with up and $down$, $current$ reifies the current level, and $lambda_at$ creates a *level-capturing function* bound at level l. When such a function is applied, execution jumps to level l and then goes back to the level prior to the application [41].

The semantics of execution levels can be embedded in the definition of aspects themselves, by transforming the pointcut and advice of an aspect at deployment time, as shown in Fig. 9.[20] This is done by functions $pcEL$ and $advEL$. $pcEL$ first ensures that the current execution level $lapp$ matches $ldep$, the level at which the aspect is deployed. If so it then runs the pointcut one level above. Similarly, $advEL$ ensures that the advice is run one level above, with a $proceed$ function that captures the deployment level.

Example. Figure 10 defines a generic logging advice, $logAdv$, which appends the argument and result of advised functions to the log[21]. In $program$, we deploy an aspect that

[20] For simplicity, in Sect. 3.2 we only described the default semantics of aspect deployment; aspect (un)deployment is actually defined using overloaded $(un)deployInEnv$ functions.

[21] Using the $tell$ function of the $MonadWriter$ class (denoted $\mathbb{W}_\mathrm{M}$), which is not described in Sect. 2, but which essentially is a state monad with append-only access.

```
1  type Level = Int
2  newtype 𝔼𝕃T m a = 𝔼𝕃T (𝕊T Level m a)
3     -- primitive operations
4  inc  = 𝔼𝕃T $ λl → return ((), l + 1)
5  dec  = 𝔼𝕃T $ λl → return ((), l − 1)
6  at l = 𝔼𝕃T $ λ_ → return ((), l)
7     -- user-visible operations
8  current = 𝔼𝕃T $ λl → return (l, l)
9  up c    = do {inc; result ← c; dec; return result}
10 down c  = do {dec; result ← c; inc; return result}
11 lambda_at f l = λarg → do n ← current
12                           at l
13                           result ← f arg
14                           at n
15                           return result
```

Fig. 8. Execution levels monad transformer and level-shifting operations

```
deployInEnv (Aspect (pc :: PC (𝔸T (𝔼𝕃T m)) tpc) adv) aenv =
  let
    pcEL ldep = (PC $ return $ λjp → do
      lapp ← current
      if lapp ≡ ldep then up $ runPC pc jp
                     else return False) :: PC (𝔸T (𝔼𝕃T m)) tpc
    advEL ldep proceed arg = up $ adv (lambda_at proceed ldep) arg
  in do l ← current
        return EAspect (Aspect (pcEL l) (advEL l)) : aenv
```

Fig. 9. Redefining aspect deployment for execution levels semantics. An aspect is made level-aware by transforming its pointcut and advice.

intercepts all calls to *showM* (the monadic version of *show*) where the argument is of type *Int* (we require a type annotation for the pointcut because *showM* is a bounded polymorphic function—see Sect. 9.2 for details).

The evaluation of the program depends on the instantiation of the monad stack M. In a setting without execution levels, advising *showM* with *logAdv* triggers an infinite loop because *logAdv* internally performs open applications of *showM*, which are matched by the same aspect. Using the execution level semantics, evaluation terminates because the join point emitted by the advice is not visible to the aspect itself.

Interestingly, explicit open applications limit the possibilities of unwanted advising. More obliviousness, *e.g.*, through partial application of #, makes it harder to track down these issues (we come back to obliviousness in Sect. 9.3). Nevertheless, identifying the source of the regression is not sufficient *per se*: in our example, if it is necessary for *logAdv* to use open applications (so that other aspects can intervene), there is not much that can be done to avoid regression.

$$
\begin{aligned}
&\mathit{showM}\ a = \mathit{return}\ (\mathit{show}\ a)\\
&\mathit{logAdv}\ \mathit{proceed}\ a = \mathbf{do}\ \mathit{argStr} \leftarrow \mathit{showM}\ \#\ a\\
&\qquad\qquad\qquad\qquad\quad \mathit{tell}\ (\texttt{"Arg: "} +\!\!+\ \mathit{argStr})\\
&\qquad\qquad\qquad\qquad\quad \mathit{result} \leftarrow \mathit{proceed}\ a\\
&\qquad\qquad\qquad\qquad\quad \mathit{return}\ \mathit{result}\\
&\mathit{program}\ n = \mathit{runM}\ \$\ \mathbf{do}\\
&\quad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ (\mathit{showM} :: \rightarrow \mathit{Int} \rightarrow M\ \mathit{String}))\ \mathit{logAdv})\\
&\quad \mathit{showM}\ \#\ n
\end{aligned}
$$

Fig. 10. A program that loops unless execution levels are used

Beyond Execution Levels. Execution levels adds a topological dimension to the composition of aspects into a system. However, their tower-like structure may be too restricted for certain scenarios, for instance for dynamic analyses aspects [43]. Recently, Tanter *et al.* proposed *programmable membranes* [44] as a generalization of execution levels. We have developed a prototype implementation of membrane semantics in Effective Aspects [12], using the same approach of converting pointcuts and advices at deployment time. However, instead of passing the current level of execution (an integer), we maintain the bindings between membranes (a graph) using a state monad.

8.5 Reasoning about Language Extensions

The above extensions can be implemented in an dynamically typed language such as LAScheme [41]. However, it is challenging to provide any kind of reasoning about effects due to the dynamic nature of the language.

Enforcing Non-interference in Language Extensions. We can combine the monadic interpretation of execution levels with the management of effect interference (Sect. 7) in order to reason about level-shifting operations performed by base and aspect computations. For instance, it becomes possible to prevent aspect and/or base computation to use effects provided by the $\mathbb{EL}_{\mathrm{T}}$ monad transformer, thus ensuring that the default semantics of execution levels is preserved (and therefore that the program is free of aspect loops [42]). For this we must consider a concrete monad stack that has the $\mathbb{A}_{\mathrm{T}}$ and $\mathbb{EL}_{\mathrm{T}}$ transformers on top:

$$\mathbf{type}\ \mathbb{AEL}_{\mathrm{T}}\ m = \mathbb{A}_{\mathrm{T}}\ \mathbb{EL}_{\mathrm{T}}\ m$$

Observe that this monad stack is general with respect other effects it may contain. Then, we simply define an advice combinator that forbids access to the $\mathbb{EL}_{\mathrm{T}}$ layer, which provides the level-shifting operations.

$$\mathit{levelAgnosticAdv} = \mathit{withView}\ (\square ::: \blacksquare ::: \square)$$

This mask hides the layer with the execution-level-related effects, but allows access to $\mathbb{A}_{\mathrm{T}}$ at the top, and to the rest of the stack. Then to ensure level agnostic advice we just redefine *program* to use this combinator, in a suitable monad stack M:[22]

[22] We use the $\mathit{WriterT}$ transformer ($\mathbb{W}_{\mathrm{T}}$), which is the canonical instance of $\mathbb{W}_{\mathrm{M}}$.

$$
\begin{aligned}
&\mathbf{type}\ M = \mathbb{AEL}_{\mathrm{T}}\ (\mathbb{W}_{\mathrm{T}}\ \mathit{String}\ \mathbb{I})\\
&\mathit{runM}\ c\ = \mathit{run}\mathbb{I}\ \$\ \mathit{run}\mathbb{W}_{\mathrm{T}}\ \$\ \mathit{run}\mathbb{EL}_{\mathrm{T}}\ (\mathit{run}\mathbb{A}_{\mathrm{T}}\ c)\ 0\\
&\mathit{program}\ n = \mathit{runM}\ \$\ \mathbf{do}\\
&\quad \mathit{deploy}\ (\mathit{aspect}\ (\mathit{pcCall}\ (\mathit{showM} :: \to \mathit{Int} \to M\ \mathit{String}))\\
&\qquad\qquad\qquad (\mathit{levelAgnosticAdv}\ \mathit{logAdv}))\\
&\quad \mathit{showM}\ \#\ n
\end{aligned}
$$

If more advanced use of execution levels is required, this contraint can be explicitly relaxed in the $\mathbb{A}_{\mathrm{T}}$ or $\mathbb{EL}_{\mathrm{T}}$ monad transformer, thus stressing in the type that it is the responsibility of the programmer to avoid infinite regression.

Using Types to Enforce Weaving Semantics. The type system makes it possible to specify functions that can be woven, but only within a specific aspect monad. For instance, suppose that we want to define a *critical* computation, which must only be run with secure weaving for access control. The computation must therefore be run within the $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$ monad transformer with a given pointcut pc_ac (ac stands for access control).

To enforce the use of $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$ with a specific pointcut value would require the use of a dependent type, which is not possible in Haskell. This said, we can use the **newtype** data constructor together with its ability to derive automatically type class instances, to define a new type $\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}$ that encapsulates the $\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}$ monad transformer and forces it to be run with the pc_ac pointcut:

$$
\begin{aligned}
&\mathbf{newtype}\ \mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ m\ a = \mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ m\ a)\ \mathbf{deriving}\ (\mathit{Monad}, \mathit{OpenApp}, \ldots)\\
&\mathit{runSafe}\ (\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ c) = \mathit{run}\mathbb{A}_{\mathrm{T}}^{\mathrm{S}}\ (\mathit{EPC}\ \mathit{pc_ac})\ c
\end{aligned}
$$

Therefore, we can export the *critical* computation by typing it appropriately:

$$
\mathit{critical} :: \mathit{Monad}\ m \Rightarrow \mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}\ m\ a
$$

Because the $\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}$ constructor is hidden in a module, the only way to run such a computation typed as $\mathbb{A}_{\mathrm{T}}^{\mathrm{AC}}$ is to use *runSafe*. The *critical* computation is then only advisable with secure weaving for access control.

9 Discussion

We now discuss a number of issues related to our approach: how to define a proper notion of function equality, how to deal with overloaded functions, and finally, we analyze the issue of obliviousness.

9.1 Supporting Equality on Functions

Pointcuts quantify about join points, and a major element of the join point is the function being applied. The *pcType* designator relies on type comparison, implemented using the *PolyTypeable* type class in order to obtain representations for polymorphic types. The *pcCall* is more problematic, as it relies on function equality, but Haskell does not provide an operator like *eq?* in Scheme.

A first workaround is to use the $StableNames$ API that allows comparing functions using pointer equality. Unfortunately, this notion of equality is fragile. $StableNames$ equality is safe in the sense that it does not equate two functions that are not the same, but two functions that are equal can be seen as different.

The problem becomes even more systematic when it comes to bounded polymorphism. Indeed, each time a function with constraints is used, a new closure is created by passing the current method dictionary of type class instances. Even with optimized compilation (*e.g.*, ghc -O), this (duplicated) closure creation is unavoidable and so $StableNames$ will consider different any two constrained functions, even if the passed dictionary is the same.

To overcome this issue, we have overloaded our equality on functions with a special case for functions that have been explicitly tagged with a unique identifier at creation (using $Data.Unique$). This allows us to have a robust notion of function equality but it has to be used explicitly at each function definition site.

9.2 Advising Overloaded Functions

From a programmer's point of view, it can be interesting to advise an overloaded function (that is, the application of all the possible implementations) with a single aspect. However, deploying aspects in the general case of bounded polymorphism is problematic because of the resolution of class constraints. Recall that in order to be able to type the aspect environment, we existentially hide the matched and advised types of an aspect. This means that all type class constraints must be solved statically at the point an aspect is deployed. If the matched and advised types are both bounded polymorphic types, type inference cannot gather enough information to statically solve the constraints. So advising all possible implementations requires repeating deployment of the same aspect with different type annotations, one for each instance of the involved type classes.

To alleviate this problem, we developed a macro using TemplateHaskell [36]. The macro extracts all the constrained variables in the matched type of the pointcut, and generates an annotated deployment for every possible combination of instances that satisfy all constraints. In order to retain safety, the advised type of an aspect must be less constrained than its matched type. This is statically enforced by the Haskell type system after macro expansion.

9.3 Obliviousness

The embedding of aspects we have presented thus far supports quantification through pointcuts, but is not oblivious: open applications are explicit in the code. A first way to introduce more obliviousness without requiring non-local macros or, equivalently, a preprocessor or ad hoc runtime semantics, is to use *partial applications* of $\#$. For instance, the $enqueue$ function can be turned into an implicitly woven function by defining $enqueue' = enqueue\ \#$. This approach was used in Fig. 2 for the definition of fib. It can be sufficient in similar scenarios where quantification is under control. Otherwise, it can yield issues in the definition of pointcuts that rely on function identity, because $enqueue'$ and $enqueue$ are different functions. Also, this approach is not entirely

satisfactory with respect to obliviousness because it has to be applied specifically for each function.

De Meuter proposes [26] to use the binder of a monad to redefine function application. His approach focuses on defining one monad per aspect, but can be generalized to a list of dynamically deployed aspects as presented in Sect. 3.2. For this, we can redefine the monad transformer $\mathbb{A}_{\mathrm{T}}$ to make all monadic applications open transparently:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (\mathbb{A}_{\mathrm{T}}\ m)\ \textbf{where} \\
\quad \mathit{return}\ a = \mathbb{A}_{\mathrm{T}}\ \$\ \lambda \mathit{aenv} \rightarrow \mathit{return}\ (a, \mathit{aenv}) \\
\quad k \gg\!= f \quad = \textbf{do}\ x \leftarrow k \\
\qquad\qquad\qquad\quad f \mathbin{\#} x
\end{array}
$$

This presentation improves obliviousness because any monadic application is now an open application, but it suffers from a major drawback: it breaks the *monadic laws*. Indeed, left identity and associativity

$$
\begin{array}{l}
\text{-- Left identity:} \\
\mathit{return}\ x \gg\!= f = f\ x \\
\text{-- Associativity:} \\
(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)
\end{array}
$$

can be invalidated, depending on the current list of deployed aspects. This is not surprising as AOP allows one to redefine the behavior of a function and even to redefine the behavior of a function depending on its context of execution. Breaking monadic laws is not prohibited by Haskell, but it is very dangerous and fragile; for instance, some compilers exploit the laws to perform optimizations, so breaking them can yield incorrect optimizations.

9.4 Technical Requirements of Our Model

The current implementation of Effective Aspect uses several extensions of the GHC Haskell compiler (see the details at `http://plead.cl/effectiveaspects`). Nevertheless, we believe that the anti-unification algorithm at the type level (Section 4.1) is the essential feature that would be required to make our approach work on other languages. A potential line of work is to port Effective Aspects to Scala, which has some likeness to Haskell and also has monads, and investigate what kind of issues arise in the process.

10 Related Work

The earliest connection between aspects and monads was established by De Meuter in 1997 [26]. In that work, he proposes to describe the weaving of a given aspect directly in the binder of a monad. As we have just described above (Sect. 9.3), doing so breaks the monad laws, and is therefore undesirable.

Wand *et al.* [48] formalize pointcuts and advice and use monads to structure the denotational semantics; a monad is used to pass the join point stack and the store around evaluation steps. The specific flavor of AOP that is described is similar to AspectJ, but with only pure pointcuts. The calculus is untyped. The reader may have noticed that we do not model the join point stack in this paper. This is because it is not *required* for a given model of AOP to work. In fact, the join point stack is useful only to express control flow pointcuts. In our approach, this is achieved by specifying a user-defined pointcut designator for control flow, which uses a monad to thread the join point stack (or, depending on the desired level of dynamicity, a simple control flow state [24]). Support for the join point stack does not have to be included as a primitive in the core language. This is in fact how AspectJ is implemented [24,15].

Hofer and Osterman [16] shed some light on the modularity benefits of monads and aspects, clarifying that they are different mechanisms with quite different features: monads do not support declarative quantification, and aspects do not provide any support for encapsulating computational effects. In this regard, our work does not attempt at unifying monads and aspects, contrary to what De Meuter suggested. Instead, we exploit monads in Haskell to build a flexible embedding of aspects that can be modularly extended. In addition, the fully typed setting provides the basis for reasoning about monadic effects.

The notion of *monadic weaving* was described by Tabareau [38], where he shows that writing the aspect weaver in a monadic style paves the way for modular language extensions. He illustrated the extensibility approach with execution levels [41] and level-aware exception handling [13]. The authors then worked on a practical monadic aspect weaver in Typed Racket [14]. However, the type system of Typed Racket turned out to be insufficiently expressive, and the top type Any had to be used to describe pointcuts and advices. This was the original motivation to study monadic weaving in Haskell. Also in contrast to this work, prior work on monadic aspect weaving does not consider a base language with monads. In this paper, both the base language and the aspect weaver are monadic, combining the benefits of type-based reasoning about effects (Sect. 6) and modular language extensions (Sect. 8)—including type-based reasoning about language extensions.

Haskell has already been the subject of AOP investigations using the type class system as a way to perform static weaving [37]. AOP idioms are translated to type class instances, and type class resolution is used to perform static weaving. This work only supports simple pointcuts, pure aspects, and static weaving, and is furthermore very opaque to modular changes as the translation of AOP idioms is done internally at compile time.

The specific flavor of pointcut/advice AOP that we developed is directly inspired by AspectScheme [11] and AspectScript [45]: dynamic aspect deployment, first-class aspects, and extensible set of pointcut designators. While we have not yet developed the more advanced scoping mechanisms found in these languages [40], we believe there are no specific challenges in this regard. The key difference here is that these languages are both dynamically typed, while we have managed to reconcile this high level of flexibility with static typing.

In terms of statically typed functional aspect languages, the closest proposal to ours is AspectML [7]. In AspectML, pointcuts are first-class, but advice is not. The set of pointcut designators is fixed, as in AspectJ. AspectML does not support: advising anonymous functions, aspects of aspects, separate aspect deployment, and undeployment. AspectML was the first language in which first-class pointcuts were statically typed. The typing rules rely on anti-unification, just like we do in this paper. The major difference, though, is that AspectML is defined as a completely new language, with a specific type system and a specific core calculus. Proving type soundness is therefore very involved [7]. In contrast, we do not need to define a new type system and a new core calculus. Type soundness in our approach is derived straightforwardly from the type class that establishes the anti-unification relation. Half of section 5 is dedicated to proving that this type class is correct. Once this is done (and it is a result that is independent from AOP), proving aspect safety is direct. Another way to see this work is as a new illustration of the expressive power of the type system of Haskell, in particular how phantom types and type classes can be used in concert to statically type embedded languages.

Aspectual Caml [25] is another polymorphic aspect language. Interestingly, Aspectual Caml uses type information to influence matching, rather than for reporting type errors. More precisely, the type of pointcuts is inferred from the associated advices, and pointcuts only match join points that are valid according to these inferred types. We believe this approach can be difficult for programmers to understand, because it combines the complexities of quantification with those of type inference. Aspectual Caml is implemented by modifying the Objective Caml compiler, including modifications to the type inference mechanism. There is no proof of type soundness.

The advantages of our typed embedding do not only lie within the simplicity of the soundness proof. They can also be observed at the level of the implementation. The AspectML implementation is over 15,000 lines of ML code [7], and the Aspectual Caml implementation is around 24,000 lines of Objective Caml code [25]. In contrast, our implementation, including the execution levels extension (Sect. 8), is only 1,600 lines of Haskell code. Also, embedding an AOP extension entirely inside a mainstream language has a number of practical advantages, especially when it comes to efficiency and maintainability of the extension.

Finally, reasoning about advice effects has been studied from different angles. For instance, harmless advice can change termination behavior and use I/O, but no more [6]. A type and effect system is used to ensure conformance. Translucid contracts use grey box specifications and structural refinement in verification to reason about control effects [5]. In this work, we rather follow the type-based approach of EffectiveAdvice (EA) [28], which also accounts for various control effects and arbitrary computational effects. A limitation of EA is its lack of support quantification. A contribution of this work is to show how to extend this approach to the pointcut/advice mechanism. The subtlety lies in properly typing pointcuts. An interesting difference between both approaches is that in EA, it is not possible to talk about "the effects of all applied advices". Once an advice is composed with a base function, the result is seen as a base function for the following advice. In contrast, our approach, thanks to the aspect environment and

dynamic weaving, makes it possible to keep aspects separate and ensure base/aspect separation at the effect level even in presence of multiple aspects. We believe that this splitting of the monad stack is more consistent with programmers expectations.

11 Conclusion

We develop a novel approach to embed aspects in an existing language. We exploit monads and the Haskell-type system to define a typed monadic embedding that supports both modular language extensions and reasoning about effects with pointcut/advice aspects. We show how to ensure type soundness by design, even in presence of user-extensible pointcut designators, relying on a novel-type class for establishing anti-unification. Compared to other approaches to statically typed polymorphic aspect languages, the proposed embedding is more lightweight, expressive, extensible, and amenable to interference analysis. The approach can combine Open Modules and EffectiveAdvice, and supports type-based reasoning about modular language extensions.

Acknowledgments. This work was supported by the INRIA Associated team REAL. We thank the anonymous reviewers from the AOSD'13 conference and from this journal, and we also thank Tom Schrijvers for all his useful feedback.

References

1. In: Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012), Potsdam, Germany. ACM Press (March 2012)
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. In: Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008), Brussels, Belgium. ACM Press (April 2008)
4. In: Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010), Rennes and Saint Malo, France. ACM Press (March 2010)
5. Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.: Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In: Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011), Porto de Galinhas, Brazil. ACM Press (March 2011)
6. Dantas, D.S., Walker, D.: Harmless advice. In: Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2006), Charleston, South Carolina, pp. 383–396. ACM Press (January 2006)
7. Dantas, D.S., Walker, D., Washburn, G., Weirich, S.: AspectML: A polymorphic aspect-oriented functional programming language. ACM Transactions on Programming Languages and Systems 30(3), Article No. 14 (May 2008)
8. De Fraine, B., Südholt, M., Jonckers, V.: StrongAspectJ: flexible and safe pointcut/advice bindings. In: AOSD 2008 [3], pp. 60–71
9. Douence, R., Fradet, P., Südholt, M.: Trace-based aspects. In: Filman, R.E., Elrad, T., Clarke, S., Akşit, M. (eds.) Aspect-Oriented Software Development, pp. 201–217. Addison-Wesley, Boston (2005)
10. Douence, R., Motelet, O., Südholt, M.: A formal definition of crosscuts. In: Matsuoka, S. (ed.) Reflection 2001. LNCS, vol. 2192, pp. 170–186. Springer, Heidelberg (2001)

11. Dutchyn, C., Tucker, D.B., Krishnamurthi, S.: Semantics and scoping of aspects in higher-order languages. Science of Computer Programming 63(3), 207–239 (2006)
12. Figueroa, I., Tabareau, N., Tanter, É.: Taming aspects with monads and membranes. In: Proceedings of the 12th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2013), Fukuoka, Japan, pp. 1–6. ACM Press (March 2013)
13. Figueroa, I., Tanter, É.: A semantics for execution levels with exceptions. In: Proceedings of the 10th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2011), Porto de Galinhas, Brazil, pp. 7–11. ACM Press (March 2011)
14. Figueroa, I., Tanter, É., Tabareau, N.: A practical monadic aspect weaver. In: Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012), pp. 21–26 (2012)
15. Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: Lieberherr, K. (ed.) Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, UK, pp. 26–35. ACM Press (March 2004)
16. Hofer, C., Ostermann, K.: On the relation of aspects and monads. In: Proceedings of AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), pp. 27–33 (2007)
17. Jones, M.P.: Type classes with functional dependencies. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 230–244. Springer, Heidelberg (2000)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of aspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
20. Learn you a haskell website (2013), `http://learnyouahaskell.com/`
21. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: Ball, T. (ed.) Proceedings of the 2nd USENIX Conference on Domain-Specific Languages, pp. 109–122 (1999)
22. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL 95), San Francisco, California, USA, pp. 333–343. ACM Press (1995)
23. Hansen, K.A., Kawauchi, K.: Dataflow pointcut in aspect-oriented programming. In: Ohori, A. (ed.) APLAS 2003. LNCS, vol. 2895, pp. 105–121. Springer, Heidelberg (2003)
24. Hansen, K.A., Kiczales, G., Dutchyn, C.: A compilation and optimization model for aspect-oriented programs. In: Hedin, G. (ed.) CC 2003. LNCS, vol. 2622, pp. 46–60. Springer, Heidelberg (2003)
25. Masuhara, H., Tatsuzawa, H., Yonezawa, A.: Aspectual Caml: an aspect-oriented functional language. In: Proceedings of the 10th ACM SIGPLAN Conference on Functional Programming (ICFP 2005), Tallin, Estonia, pp. 320–330. ACM Press (September 2005)
26. Meuter, W.D.: Monads as a theoretical foundation for aop. In: International Workshop on Aspect-Oriented Programming at ECOOP, p. 25. Springer (1997)
27. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
28. Oliveira, B.C.D.S., Schrijvers, T., Cook, W.R.: EffectiveAdvice: discplined advice with explicit effects. In: AOSD 2010 [4], pp. 109–120
29. Oliveira, B.C.D.S., Schrijvers, T., Cook, W.R.: MRI: Modular reasoning about interference in incremental programming. Journal of Functional Programming 22, 797–852 (2012)
30. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. Journal of Functional Programming 17(1), 1–82 (2007)
31. Pierce, B.C.: Types and programming languages. MIT Press, Cambridge (2002)
32. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence 5, 153–163 (1970)

33. Reynolds, J.C.: Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence 5, 135–151 (1970)
34. Rinard, M., Salcianu, A., Bugrara, S.: A classification system and analysis for aspect-oriented programs. In: Proceedings of the 12th ACM Symposium on Foundations of Software Engineering (FSE 12), pp. 147–158. ACM Press (2004)
35. Schrijvers, T., Oliveira, B.C.: Monads, zippers and views: virtualizing the monad stack. In: Proceedings of the 16th ACM SIGPLAN Conference on Functional Programming (ICFP 2011), Tokyo, Japan, pp. 32–44. ACM Press (September 2011)
36. Sheard, T., Jones, S.P.: Template meta-programming for haskell. SIGPLAN Not. 37(12), 60–75 (2002)
37. Sulzmann, M., Wang, M.: Aspect-oriented programming with type classes. In: Proceedings of the Sixth Workshop on Foundations of Aspect-Oriented Languages (FOAL 2007), Vancouver, British Columbia, Canada, pp. 65–74. ACM Press (2007)
38. Tabareau, N.: A monadic interpretation of execution levels and exceptions for AOP. In: Tanter, É., Sullivan, K.J. (eds.) Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD 2012), Potsdam, Germany. ACM Press (March 2012)
39. Tabareau, N., Figueroa, I., Tanter, É.: A typed monadic embedding of aspects. In: Kinzle, J. (ed.) Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013), Fukuoka, Japan, pp. 171–184. ACM Press (March 2013)
40. Tanter, É.: Expressive scoping of dynamically-deployed aspects. In: AOSD 2008 [3], pp. 168–179
41. Tanter, É.: Execution levels for aspect-oriented programming. In: AOSD 2010 [4], pp. 37–48
42. Tanter, É., Figueroa, I., Tabareau, N.: Execution levels for aspect-oriented programming: Design, semantics, implementations and applications. Science of Computer Programming (2013) (available online)
43. Tanter, É., Moret, P., Binder, W., Ansaloni, D.: Composition of dynamic analysis aspects. In: Proceedings of the 9th ACM SIGPLAN International Conference on Generative Programming and Component Engineering (GPCE 2010), Eindhoven, The Netherlands, pp. 113–122. ACM Press (October 2010)
44. Tanter, É., Tabareau, N., Douence, R.: Taming aspects with membranes. In: Proceedings of the 11th Workshop on Foundations of Aspect-Oriented Languages (FOAL 2012) [1], pp. 3–8
45. Toledo, R., Leger, P., Tanter, É.: AspectScript: Expressive aspects for the Web. In: AOSD 2010 [4], pp. 13–24
46. Wadler, P.: The essence of functional programming. In: Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL 1992), Albuquerque, New, Mexico, USA, pp. 1–14. ACM Press (January 1992)
47. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL 1989), Austin, TX, USA, pp. 60–76. ACM Press (January 1989)
48. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. ACM Transactions on Programming Languages and Systems 26(5), 890–910 (2004)

Modular Specification and Checking of Structural Dependencies

Ralf Mitschke[1], Michael Eichberg[1], Mira Mezini[1], Alessandro Garcia[2], and Isela Macia[2]

[1] Technische Universität Darmstadt
{mitschke,eichberg,mezini}@informatik.tu-darmstadt.de
[2] Pontifical Catholic University of Rio de Janeiro
{afgarcia,ibertran}@inf.puc-rio.br

Abstract. Checking a software's structural dependencies is a line of research on methods and tools for analyzing, modeling, and checking the conformance of source code w.r.t. specifications of its intended static structure. Existing approaches have focused on the correctness of the specification, the impact of the approaches on software quality and the expressiveness of the modeling languages. However, large specifications become unmaintainable in the event of evolution without the means to modularize such specifications. We present Vespucci, a novel approach and tool that partitions a specification of the expected and allowed dependencies into a set of cohesive slices. This facilitates modular reasoning and helps individual maintenance of each slice. Our approach is suited for modeling high-level as well as detailed low-level decisions related to the static structure and combines both in a single modeling formalism. To evaluate our approach, we conducted an extensive study spanning 9 years of the evolution of the architecture of the object-relational mapping framework Hibernate.

Keywords: Software Architectures, Modularity, Scalability, Structural Dependency Constraints, Static Analysis.

1 Introduction

A documented software architecture is an acknowledged success factor for the development of large, complex systems [1]. Traditionally, architecture description languages (ADLs) have been used to specify the architecture and verify its properties. Generally, this process has been detached from coding. The architecture specification has been considered as a means to prescribe the structure of the code resulting from programming or to eventually generate a first skeleton of that code. However, as systems evolve over time, due to new requirements or corrections, the implemented architecture starts to diverge from the intended architecture [2–4] — resulting in *architecture erosion* [5].

To combat architecture erosion, several approaches have emerged that focus on structural dependencies [6–9] and whose proponents argue for automated checking of architecture specifications w.r.t. the static structure of the source code.

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 193–226, 2014.
© Springer-Verlag Berlin Heidelberg 2014

These approaches generally allow to group[1] source code elements into *building blocks* — cohesive units of functionality in the software system — and to specify in which way a building block is allowed to statically depend on which other building block. The specification formalisms in these approaches vary and can be summarized as: (i) a flat graph with building blocks as nodes and allowed dependencies as edges [7]; (ii) a matrix notation with building blocks in rows/-columns and their dependencies in the cells [8]; (iii) a graph with hierarchical nodes and component-connector style ports to manage internal/external dependencies [6]; (iv) a textual specification of access restrictions on target building blocks [9].

Such specifications are used either analytically [7] — to analyze already written code for conformance with an intended static structure — or constructively [6, 9] to enforce the code's compliance with the specification of the static structure continuously during development. Constructive approaches were proven to help developers in realizing the intended architecture. Several case studies [10–12] show that constructive approaches can prevent structural erosion [13, 14].

Though current approaches have proven to be valuable, they all share the property that a single monolithic specification is used and – as in case of a monolithic software system — a monolithic description of the structure does not scale and becomes unmaintainable once the software reaches a certain complexity. Sangal et al. [8] explicitly try to solve the maintainability and scalability issues using a special notation called dependency structure matrices (DSMs). However, we believe that the problem is not so much the notation. The root of the problem is the monolithic nature of the specifications. Based on some preliminary experience with modeling the architecture of real systems, such as Hibernate [15], we doubt that any such approach can scale, even with compact notations such as dependency structure matrices. As a result, typically only the highest level of components and/or libraries is considered [9, 13]; requiring different notations and tools for different levels of the design. This precludes a seamless design at various granularity levels.

In this paper, we argue that modeling a software's static structure should consist of multiple views, that focus on different parts and on different levels of detail. We take the position that like programming languages, architecture modeling languages in general should support modularity and scoping mechanisms to support modular reasoning about different architectural concerns and information hiding to facilitate evolution.

Accordingly, we propose a novel modeling approach and tool, called Vespucci, that allows to separate the specification of a software's static structure into multiple complementary views, called *slices* throughout this paper. Each slice can be reasoned over in separation. Multiple slices can express different views on the same part of the software and each slice can be evolved individually. Hence, evolution of large scale specifications consisting of several slices is facilitated by distributing work to systematically update the architecture in a modular fashion. The proposed approach is at least as expressive as existing models for structural

[1] Using, e.g., regular expressions over classes or source files.

rules. Hence, the strong utility of these rules (as shown in [13, 14]) is retained. Yet, we enhance the modularity of the specification language. Contrary to a monolithic specification of previous models, our approach has the benefit that individual concerns can remain stable. Stable parts can be modularized into different slices to be separated from architectural "hot-spots," i.e., slices that require frequent changes during the evolution.

The contributions of this paper are:

– A first approach toward the specification of a software's structural dependencies that supports a modularized specification by means of individual slices.
– A new approach for modeling a software's structural dependencies that combines the advantages of hierarchical and graph-based modeling approaches to enable reasoning over a software's static structure at different abstraction levels.
– Discussion of an implementation of the proposed approach that enables the specification and checking of a software's structural dependencies.

The remainder of the paper is organized as follows. In Sec. 2, we briefly introduce the Hibernate framework [15], which we use to illustrate concepts of the proposed approach and to evaluate its effectiveness. Section 3 introduces Vespucci's specification language. In Sec. 4 we present an in-depth evaluation of Vespucci. After that, we discuss related work in Sec. 5. Finally, we give a summary and discuss future work.

2 Architecture of Hibernate

As part of the development of Vespucci, we did a comprehensive analysis of the architecture of the object-relational mapping framework Hibernate [15]. We provide a short overview of Hibernate and its architecture in this section since we will refer to it to discuss and motivate various features of Vespucci.

We chose Hibernate as it is a large, mature, widely adopted software system, which has been continually updated and enhanced. We reengineered the architecture of the core of Hibernate in version 1.0.1 (July 2002) and played back its evolution until version 3.6.6 (July 2011)[2]. During this time the core grew from 2 000 methods in over 255 classes organized in 18 packages to 17 700 methods in over 1 954 classes in 100 packages.

In the following, the major building blocks of Hibernate's architecture are presented. A *building block* is a logical grouping of source code elements that provide a cohesive functionality, independent of the program's structuring, e.g., in packages or classes. The scope of a building block depends on the considered abstraction level and ranges from a few source code elements up to several hundreds. For example, Hibernate's support for different SQL dialects is represented by one top-level building block with many source code elements that is further structured into smaller building blocks for elements that abstract over the support for concrete dialects and those that actually implement the support.

[2] Hibernate 4.0 was released after the case study.

Table 1. Overview of Hibernate 1.0

Top-level Building Block	2L Building Blocks	Classes contained	Elements contained	Relation to Packages
Cache	4	6	60	$\equiv$
CodeGeneratorTool	0	9	68	$\subset$
ConnectionProvider	3	5	51	$\equiv$
DatabaseActions	3	9	59	$\subset$
DataTypes	10	37	410	$\equiv$
DeprecatedLegacy	2	2	6	$\subset$
EJBSupport	0	1	22	$\equiv$
HibernateORConfiguration	2	2	39	$\times$
HibernateORMapping	12	33	389	$\equiv$
HQL (Hibernate Query Language)	3	9	130	$\equiv$
IdentifierGenerators	4	12	92	$\equiv$
MappingGeneratorTool	0	19	233	$\subset$
PersistenceManagement	6	35	674	$\times$
PropertySettings	0	1	43	$\times$
Proxies	0	3	23	$\subset$
SchemaTool	2	5	34	$\subset$
SessionManagement	6	10	312	$\times$
SQLDialects	3	12	119	$\equiv$
Transactions	2	4	37	$\equiv$
UserAPI	9	9	63	$\times$
UtilitiesAndExceptions	2	33	235	$\times$
XMLDatabinder	0	2	21	$\times$

The architectural model of Hibernate 1.0 consists of the 22 top-level building blocks shown in Table 1. Of these 22 top-level building blocks, 16 were further structured. In total, we identified 73 second-level building blocks. Given the size of Hibernate 1.0, we did not analyze lower levels. On average each top-level building block already only contains 11 classes and the 2nd level building blocks consist of even fewer classes. The key figures of the architecture are given in Table 1. In the following, we discuss those elements of the architecture that are most relevant when considering the modeling of architectures. The complete architecture can be downloaded from the project's website [16].

For Hibernate 1.0 nine of the building blocks have a one-to-one mapping to a package (cf. Table 1 – Relation to Packages $\equiv$). Six building blocks map to a subset ($\subset$) of the code of some non-cohesive package. For example, the package `cirrus.hibernate.impl` contains classes for creating proxies as well as classes related to database actions. These sets of classes have no interdependencies and belong to different building blocks. The source code elements of the remaining building blocks are spread across several packages ($\times$). For example, the code related to session handling is spread across two packages in version 1.0.

Overall, the architecture features several well modularized building blocks, such as the Cache, HQL or Transactions building blocks, which are only coupled with at most three other building blocks. The number of well-modularized building blocks with few dependencies is, however, small. The majority of Hibernate's functionality belongs to building blocks that exhibit high coupling, such as PersistenceManagement, SessionManagement, and DataTypes.

3 The Vespucci Approach

In this section, we first describe the four major parts Vespucci [16] consists of: (1) a declarative source code query language to overlay high-level abstractions over the source code, (2) an approach that enables the modular, evolvable, and scalable modeling of an application's structural dependencies, and (3) a formalism as well as (4) a tool for checking the consistency between the modeled and the implemented dependencies. After that, we present the different modeling approaches supported by Vespucci. Finally, we discuss how the proposed approach facilitates the evolution of the specification and the underlying software and how it supports large(r) scale software systems.

3.1 High-Level Abstractions over Source Code

Vespucci is concerned with modeling and controlling structural dependencies at the code level. But, it does so at a high-level of abstraction.

Ensembles are Vespucci's representation of high-level building blocks of an application whose structural dependencies are modeled and checked. Specifically, Vespucci's ensembles are groups of source code elements, namely type, method, and field declarations. The definition of an ensemble involves the specification of source code elements that belong to it by means of source code queries. We refer to the set of source code elements that belong to an ensemble as the *ensemble's extent.*

The visual notation of an ensemble is a box with a name label. For example, Fig. 1 shows two ensembles, one called SessionManagement and one called HQL. Vespucci explicitly predefines the so-called *empty ensemble* that never matches any source elements and is depicted using a simple gray box (■). The empty ensemble supports some common modeling tasks, e.g., to express that a utility package should not have any dependencies on the rest of the application's code.

The source code query language is only introduced by example in the following paragraphs, since it is not the primary focus of this paper, which is rather on modularity mechanisms for modeling structural dependencies. In fact, the approach as a whole is parameterized by the query language, in the sense that the modularization mechanisms can be reused with other more expressive query languages and more sophisticated query engines. For a more systematic definition of the current query language, the interested reader is referred to the website of the project [16].

The query language provides a set of basic predicates that can select individual fields or methods, entire classes, packages, or source files. Predicates take

quoted parameters, which filter respective code elements by their signature, e.g., the predicate `package('cirrus.hibernate.helpers')` selects code elements in Hibernate's `helpers` package, using the package name as the filter. The query defines the Utilities ensemble, which we have used in modeling Hibernate's structural dependencies.

In the above example, source code elements are precisely specified by their fully qualified signature. Furthermore, wildcards ("*") can be used to abstract over individual predicate parameters. For example, the `field` predicate below selects fields with an arbitrary name (the second parameter is "*") that are declared in classes with the simple name `Hibernate` and where the name of the field's type ends with the suffix `Type`. We have used the query to define an ensemble called TypeFactory which serves as a factory for Hibernate's built-in types.

```
field('*.Hibernate','*','*Type')
```

Queries can be composed using the standard set theoretic operations (union, intersection, difference), or by passing a query as an argument to a type parameter of another query. This form of composition is useful to reason over inheritance for selecting all sub-/supertypes of a given type. For example, consider the query:

```
class_with_members(subtype+('Dialect'))
```

It uses the basic predicate `class_with_members`, which selects a class and all it's members. Since the predicate expects a type to be selected, we can instead pass a sub-query. The `subtype+` query returns the transitive closure of all subtypes of the class `Dialect`. Hence, the example query selects all classes (and their members) that are a subtype of the class `Dialect`. In Hibernate these represent all supported SQL dialects – the shown query actually defines the ensemble ConcreteDialects.

As already mentioned, the query language is interchangeable. What is interesting about the use of the source query language as an ingredient of our approach is that it enables modeling structural dependencies at a high-level of abstraction. Furthermore, it supports the definition of ensembles that cut across the modular structure of the code, e.g., `TypeFactory` cuts across the class-based decomposition of code. This enables feature-based control of structural dependencies.

Vespucci provides an **ensemble repository** that stores the definitions of all ensembles. It serves as a project-wide repository and provides the starting point for modeling an application's intended structural dependencies. Capturing all ensemble definitions in a single repository serves two purposes. First, it enables a model of intended structural dependencies to be modularized with the guarantee that all modules refer to the same extension for a particular ensemble. Second, it allows modules to pose global constraints quantifying over all defined ensembles (see the discussion about global constraints in the following section).

3.2 Modeling Structural Dependencies

Dependency slices are Vespucci's mechanism to support the modularized specification of an application's structural dependencies. A slice captures one or more specific design decisions, by expressing one or more constraints over ensemble interdependencies, e.g., which ensemble(s) is (are) allowed to use a certain other ensemble.

For illustration, Fig. 1 shows an exemplary slice, which governs dependencies to source code elements that implement the Hibernate query language, represented by the HQL ensemble. Specifically, it states that elements pertaining to HQL may be used ONLY by those pertaining to the SessionManagement ensemble. The circle attached to the arrow pointing to HQL states this as a global property, i.e., for all ensembles in the ensemble repository, this is the only dependency on HQL's elements that is allowed.

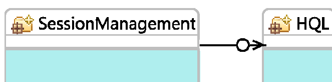


Fig. 1. Dependency Rule for Hibernate Query Language

Figure 2 shows another example slice that states that source code elements pertaining to SQLDialects are only allowed to be used by PersistenceManagement's or SessionsManagement's source code elements.

Fig. 2. Users of SQL Dialects

There can be an arbitrary number of slices in a model of structural dependencies and the set of ensembles referred to in different slices may also overlap. Deciding about the number/kind of slices, in which one breaks down the specification of an application's structural dependencies is a matter of modeling methodology, as we elaborate on in section 3.5. Yet, we envision the default strategy to be one in which each slice is used to express allowed and expected dependencies from the perspective of a single ensemble; this strategy was successfully used in the majority of the slices defined in our case study and is shown in most of the examples found in this paper. For the purpose of reasoning over which dependencies are (not) allowed for an ensemble, the visual notation features arrow symbols that are shown next to the ensemble that is constrained[3].

[3] For this paper the visual models were compressed to save space. Hence the distinction may not be as obvious as it is when you use the Vespucci tool.

For example, by looking at Fig. 1 we can reason about *all* dependencies that are allowed for HQL and looking at Fig. 2 we can reason about *all* dependencies that are allowed for SQLDialects.

Ensembles that participate in a slice but which have no arrow symbols next to their box are not constrained. For example, both slices refer to SessionManagement, but make no statement w.r.t. its allowed dependencies. From these two slices we can see that SessionManagement's source code elements are allowed to depend on both SQLDialect's and HQL's source code elements. However, SessionManagement and PersistenceManagement are not constrained.

Constraint types are classified into two basic categories: constraints that are defined w.r.t. the allowed and those w.r.t. the not-allowed dependencies. Constraints on allowed dependencies are further classified as *Outgoing and Incoming Constraints* and *Local and Global Constraints*. In addition Vespucci allows to formulate *expected* dependencies, i.e., dependencies whose presence architects actually require in the source code. The rationale for distinguishing between the above types of constraints relates to enabling modular reasoning about individual architectural concerns. Modular reasoning fosters scalability by allowing each slice to be understood as a single unit of comprehension, and also fosters evolvability as each slice can be adapted without the need to refer to other slices. We elaborate on the role that different constraint types play with these respects in the following section. Here, we exclusively focus on explaining the meaning of these different constraints.

An *incoming constraint* restricts the set of source code elements that may use the elements of a particular ensemble (target ensemble). Incoming constraints are denoted by the symbol "→" shown next to the target ensemble (cf. Fig. 1, Fig. 2). For example, the constraint in Fig. 1 restricts source code dependencies where the target element belongs to HQL: the source of the dependency must belong to SessionManagement; source code dependencies from and to the source code elements belonging to SessionManagement are — w.r.t. that slice — unrestricted.

An *outgoing constraint* restricts the set of source code elements on which code elements of a specific ensemble (source ensemble) may depend. Outgoing constraints are visually denoted by the symbol "⊳" shown next to the source ensemble. For example, the slice in Fig. 3 features two outgoing constraints; from ConnectionProvider to PropertySettings, respectively to UtilitiesAndExceptions. Outgoing constraints only affect code elements of their source ensemble. Hence, the slice in Fig. 3 governs the dependencies of code elements involved in providing connections (captured by the ConnectionProvider ensemble). They may only use generic functionality (captured by the UtilitiesAndExceptions ensemble), or functionality for getting and setting properties (captured by PropertySettings). The targets of the constraint (PropertySettings and UtilititiesAndExceptions) can — w.r.t. the slice in Fig. 3 — depend on any other ensemble.

Global constraints quantify over all defined ensembles. Visually they are denoted by a "○" attached to a constraint. All constraints considered in the examples so far were global. For example, the constraint shown in Fig. 1 affects code elements that belong to any ensemble defined in the repository of Hibernate,

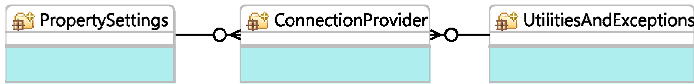


Fig. 3. Constraints on the Connection Provider

even if not referred to by the slice, e.g., ConnectionProvider or PropertySettings in Fig. 3. Code elements of the latter ensembles are not allowed to depend on elements in HQL.

Global constraints are hard constraints w.r.t. the addition of new ensembles into the architecture. Whenever new ensembles are defined in the ensemble repository, they are included when checking a global constraint. The purpose is to provide tight control over the evolution of some part of the overall architecture. If a new ensemble has dependencies that violate a global constraint, then architects can asses whether the violation needs to be removed from the code or, whether the currently defined architectural rules are too narrow. The essential point is that an architect has assessed the situation and no uncontrolled erosion of the software's structure occurs.

Local constraints quantify only over ensembles that are referenced in one particular slice. Visually, they are characterized by the lack of the "∘" symbol. Figure 4 depicts local constraints on the implementation of Hibernate's support for different SQL dialects (e.g., "Oracle SQL","DB2 SQL"). Each dialect is realized by implementing a common interface. Elements of this interface are captured by the AbstractDialect ensemble. Support for specific dialects is captured by ConcreteDialects. The TypeNameMap ensemble captures code elements involved in implementing a specialized dictionary for mapping database type names to a common set of names. The defined constraints specify that only code pertaining to ConcreteDialects is allowed to depend on code pertaining to AbstractDialect and code in the latter is only allowed to depend on TypeNameMap's code. Furthermore, neither source code elements of AbstractDialect nor TypeNameMap are allowed to depend on elements of ConcreteDialects due to the incoming constraint between the empty ensemble and ConcreteDialects. However, the constraints of the slice in Fig. 4 do not restrict in any way code elements belonging to ensembles that are not referenced by this slice, e.g., code pertaining to HQL (slice in Fig. 1) could use code pertaining to ConcreteDialects.

Local constraints provide tight control over the evolution of source code w.r.t. the scope of the ensembles referenced in a slice. Their purpose is to capture localized rules that reason only over a part of all the dependencies in the architecture, e.g., as in Fig. 4 where only dependencies pertaining to the implementation of multiple SQL dialects are considered. The implementation details of involved ensembles can change (and respectively their extensions), but the changes are guaranteed to adhere to the specified allowed/expected dependencies. The rest of the architecture can evolve independently, i.e., new ensembles and dependencies can be introduced as long as they do not violate the localized rules.

Expected constraints communicate that some dependencies in the source code must exist and guarantee the consistency w.r.t. what is actually used. For example,

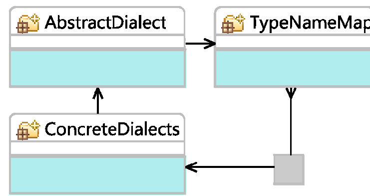


Fig. 4. Supporting Multiple SQL Dialects

Hibernate uses a logging API from the Apache project. Logging is a crosscutting concern and, hence, the library is allowed to be used by any ensemble. However, we can communicate that at least ensembles that provide core functionality, such as PersistenceManagement or SessionManagement must use the logging. If any of these ensembles stop using the logging API, Vespucci marks the respective ensembles as suspicious. The actual problems then require careful investigation, e.g., is logging not used at all or is a different logging API used. Furthermore, we treat dependencies as expected for ensembles that are explicitly allowed to use one another due to a local/global incoming or outgoing constraint. For example, in Fig. 2 we have explicitly allowed PersistenceManagement to use SQLDialects. Hence, if there is no concrete dependency in PersistenceManagement's code to SQLDialects' code the constraint is suspicious and can either be removed, or indicates further problems in the code base.

Different kinds of dependencies can be constrained individually by annotating constraints. The kinds of dependencies are those that can be found in Java code (e.g., Field Read Access, Field Write Access, Inherits, Calls, Creates,...; a complete reference is available online [16]); by default, all kinds of dependencies are constrained and no further annotation of a constraint is necessary. Dependency kinds are important when documenting detailed design choices.

For example, Fig. 5 restricts only dependencies of the create kind (i.e., object creations) to the ConcreteConnectionProviders; only the ConnectionProviderFactory is allowed to create new connection providers. All other dependencies are allowed for all ensembles, hence clients may use the created provider, e.g., by calling its methods. The range of possible applications is broad, e.g., one can also disallow classes from throwing particular exceptions, while allowing their methods to catch them.

Nesting of ensembles is also enabled in Vespucci to reflect part-whole relationships. The information about child/parent relationships between ensembles is stored in the global repository. For illustration consider that the slice shown in Fig. 4 actually models the internal architecture of Hibernate's support for SQL

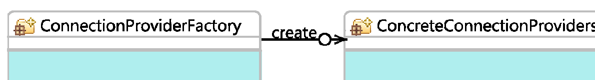


Fig. 5. Restricting Connection Provider Creation to a Factory

Fig. 6. (Sub-)Ensembles of SQL Dialects

dialects. One can express this relation by making the ensembles referred to in Fig. 4 children of the SQLDialects ensemble, as shown in Fig. 6.

The extension of an ensemble that has inner ensembles is the union of the extension of its inner ensembles; i.e., an ensemble with inner ensembles does not define its own query to match source elements, but instead reuses the queries of its inner ensembles. Hence, the semantics of nesting is that constraints defined for parent ensembles implicitly apply to source code elements of all their children, e.g., constraints defined for SQLDialects in the slice in Fig. 2 apply to all its children ensembles.

Constraints that cross an ensemble's border are disabled in Vespucci for keeping the semantics simple. Due to slices, this can be done without loss of expressivity. If an architect needs to define a constraint between two ensembles that do not have the same ancestor ensemble, it is always possible to specify the constraint in a new slice that just refers to the directly relevant ensembles.

With hierarchical modeling, architects can distinguish between ensembles that are involved in the architectural-level modeling of dependencies (SQLDialects) and those involved in modeling decisions at lower design levels (ensembles in Fig. 4). In the following sections we discuss how the combination of slices and hierarchies facilitates the incremental refinement of a software's architecture and is advantageous in case of software evolution.

3.3 Constraint Enforcement and Formal Semantics

In the following, we first describe – at the conceptual level – how the structural dependencies in the source code are checked against the modeled dependencies in Vespucci. Afterwards, we present the formal semantics of the Vespucci model and discuss several underlying design decisions.

Conceptually, checking of structural dependencies consists of the following two major steps: First, for each ensemble its extension is calculated together with the set of dependencies related to its extension. Self-dependencies, i.e., source code dependencies, where the source and target elements belong to the same ensemble are filtered out. Furthermore, dependencies from and to source code elements that do not belong to any ensemble are ignored. The calculated ensemble extensions and the actual dependencies in the source code are then used to calculate the set of dependencies between ensembles, i.e., to lift the original source code dependencies to the level of ensembles. Note that lifting the dependencies to the level of ensembles yields a clean and simple semantics, while — effectively — the dependencies between source code elements are checked.

The rationale behind the decision to ignore dependencies to source code elements that do not belong to any ensemble is that dependencies to an application's essential libraries and frameworks are most often not of architectural relevance and should not clutter the overall specification. Nevertheless, it is always possible to create an ensemble that covers some fragment of a fundamental library to restrict its usage. E.g., while it generally does not make sense to restrict the usage of the JDK, it may still be useful to restrict the usage of some API, such as the `java.util.logging` API, because the project as a whole uses a different API for logging and it has to be made sure that no one accidentally uses the default logging API. One possibility to model such a decision is to create a global incoming constraint from an empty ensemble to the ensemble representing the `java.util.logging` API.

The second step when checking structural dependencies is to perform the actual checking of the defined dependency constraints. To facilitate checking and reasoning, the underlying model is defined such that it facilitates modular architecture specifications. This means that each slice can be checked in isolation without considering constraints of other slices. To check a slice, Vespucci iterates over all ensembles of the slice and verifies that no existing dependency between two ensembles violates a defined constraint. For example, to check the compliance of an application's source code with the slice in Fig. 3, Vespucci effectively checks that the target of all code dependencies, starting at a code element in ConnectionProvider, either belongs to PropertySettings or to UtilitiesAndExceptions.

The Vespucci model of checking slices in isolation has three important properties. First, the guarantees given in one slice are not affected by other slices. For illustration consider the the set of users of SQL-Dialects as modeled by the slice shown in Fig. 2. The original architect who defined the slice expressed the system-wide constraint that only SessionManagement and PersistenceManagement may use SQL-Dialects. Now, even if another architect would define constraints w.r.t. SQL-Dialects that are less restrictive, the constraints expressed by the first architect are still in effect since the slices are checked in isolation. Second, Vespucci's slices are open to refinements, i.e., architects can further restrict the allowed usages defined in other slices. This is for example relevant to formulate at a high level that SessionManagement and PersistenceManagement may use SQL-Dialects, yet formulate at a detailed level that SQL-Dialects has internal ensembles which may not be used. Third, it is possible that two slices define "contradicting" constraints. E.g., in one slice an ensemble A is only allowed to access ensemble B and in a second slice A is only allowed to access C. However, we are not checking for such contradictions. Indeed checking the model for contradictions is in general impossible since slices can have overlapping extents and the overlap cannot be determined by simply analyzing the queries in an automated manner. Nevertheless, it is possible to check for inconsistencies w.r.t. the target software system, i.e., by explicitly computing extents. But, the checking process will remain complex and time consuming since hierarchical refinements (as exemplified above) must also be considered. Since such a check was not relevant during our case studies we currently do not support it.

Basic Definitions for Formal Checking. The fundamental data-structures used by Vespucci are defined as follows (cf. Table 2):

The *set of source code elements* (S): It contains all classes, methods and field declarations. Each declaration is uniquely identified by its fully qualified name and signature.
For example, the method `boolean isOpen()` defined by the type `Session`, has the id: `cirrus.hibernate.Session.isOpen():boolean`

The *set of dependency kinds* (K): It contains the identifiers for all kinds of structural dependencies that can occur in Java programs, e.g., `calls`, `creates`, `inherits`, etc.

The *set of source code dependencies* (D): It contains triples consisting of two source code elements where the first element is the source of a dependency and the second element is the target of the respective dependency, the third element describes the kind of the dependency.

The *set of identifiers of all ensembles* (I): Each ensemble is uniquely identified by its fully qualified name which describes the path from the root ensemble to the current ensemble in the ensemble hierarchy. For example, the ensemble AbstractDialect is fully qualified as SQLDialects.AbstractDialect.

The *multimap for the ensemble hierarchy* (H): It stores the parent-child relationships between ensembles. For each non-root ensemble e it contains an entry (p, e) where p is the parent ensemble.

From the parent-child relationships we deduce the *set of descendants* for each ensemble. The descendants are computed as the transitive closure over H, i.e., we take H as a directed graph with edges $(e_u, e_v) \in \mathsf{H}$. The descendants of

Table 2. Definition of Ensembles, Source Code Mappings, and Dependencies

Source code elements	S
Dependency kinds	K
Dependencies between source code elements	$\mathsf{D} \subseteq \mathsf{S} \times \mathsf{S} \times \mathsf{K}$
Ensemble identifiers	I
Ensemble children	$\mathsf{H} \subseteq \mathsf{I} \times \mathsf{I}$
Ensemble descendants	$\mathsf{H}^* \subseteq \mathsf{I} \times \mathsf{I} = \{ (e_{an}, e_{des}) \mid \text{there exists a path } p \text{ from } e_{an} \text{ to } e_{des} \text{ in } \mathsf{H} \}$
Ensemble extents	$\mathsf{E} \subseteq \mathsf{I} \times \mathsf{S}$
Dependencies between ensembles	$\mathsf{D}^{\uparrow} \subseteq \mathsf{I} \times \mathsf{I} \times \mathsf{K} = \{ (e_s, e_t, k) \mid e_s, e_t \in \mathsf{I} \wedge k \in \mathsf{K} \wedge \exists (c_s, c_t, k) \in \mathsf{D} \wedge \exists (e_s, c_s) \in \mathsf{E} \wedge \exists (e_t, c_t) \in \mathsf{E} \wedge e_s \neq e_t \wedge (e_s, e_t) \notin \mathsf{H}^* \wedge (e_t, e_s) \notin \mathsf{H}^* \}$

an ensemble are then all sub-ensembles to which a *path* exists in H, where a path is a non-empty sequence $\{e_1, e_2, ..., e_n\}$ such that $(e_1, e_2) \in \mathsf{H} \wedge (e_2, e_3) \in \mathsf{H} \ldots \wedge (e_{n-1}, e_n) \in \mathsf{H}$, and all e_i in the sequence are distinct. This set is later used during dependency lifting.

For each ensemble we compute its *extent* – the set of source code elements that belong to the ensemble – by evaluating its source code query. The calculated extents are stored in the set E which contains tuples of source code ensemble identifiers and source code elements. For example, if m represents the method `isOpen`, the tuple $(SessionManagement, m)$ is a member of E.

The lifted dependencies are represented as the set $\mathsf{D}^{\uparrow}$ and are triples consisting of a source and target ensemble together with the kind of the dependency. The lifting is determined based on the found source code dependencies (D). We omit all dependencies to an ensemble's ancestors, descendants, and itself to facilitate comprehension and checking. Furthermore, no slice must contain two ensembles that are in a parent–child relationship and the extent of an ensemble that is not a leaf ensemble is always equal to the union of the extents of all child ensembles.

Checking Structural Dependencies w.r.t. Slices. In the following, we formally describe how the structural dependencies in the source code are checked against the constraints defined in the slices. Table 3 lists the sets and definitions used during the checking of slices. All slices can be uniquely identified via an id m from the set M (Architectural Slices). The checking process is based on the ensembles used in the slice m — stored in the set L_m; a subset of all ensemble identifiers — and the constraints defined in the slice m — stored in the set C_m.

A concrete constraint is then a quadruple of (i) a constraint type, e.g., local_incoming (the set T contains all types supported by Vespucci), (ii) the source ensemble and (iii) the target ensemble of the constraint (both from I) together with (iv) the kind of the dependency that is being constrained (from K). To check the structural dependencies, the constraints are first normalized such that one quadruple is created for each kind of constrained dependency, e.g., if all kinds of dependencies are not allowed (as used in Figs. 1-4) then the set C_m contains one entry for each kind of dependency, i.e., a call, field read, etc. As a concrete example, if we give the slice in Fig. 1 the identifier HQL, then C_{HQL} contains the quadruples:
(global_incoming, $\mathsf{SessionMangement}$, HQL, call),
(global_incoming, $\mathsf{SessionMangement}$, HQL, field read), etc.

From the constraints defined in a slice we deduce two sets: (i) dependencies between ensembles that are not allowed, denoted as N_m, and (ii) dependencies between ensembles that are expected, denoted as X_m. A structural violation of the dependencies found in the analyzed source code w.r.t. the slice m is then defined as V_m and determined as follows: If there exists a not-allowed dependency in N_m and the set of lifted dependencies ($\mathsf{D}^{\uparrow}$) contains the same element, a violation is identified. I.e., the dependency in the source code that resulted in the lifted dependency is a violation. Vice versa, if there exists an expected dependency in X_m and the set of lifted dependencies ($\mathsf{D}^{\uparrow}$) does not contain such a dependency we also found a violation. Note that we keep a reference to the original dependency (from D) for

Table 3. Checking Structural Consistency of Dependency Slices

Dependency slice identifiers	M
Ensembles in slice $m \in \mathsf{M}$	$\mathsf{L}_m \subseteq \mathsf{I}$
Constraint types	T
Constraints in slice $m \in \mathsf{M}$	$\mathsf{C}_m \subseteq \mathsf{T} \times \mathsf{L}_m \times \mathsf{L}_m \times \mathsf{K}$
Not allowed ensemble dependencies in slice $m \in \mathsf{M}$	$\mathsf{N}_m \subseteq \mathsf{I} \times \mathsf{I} \times \mathsf{K}$
Expected ensemble dependencies in slice $m \in \mathsf{M}$	$\mathsf{X}_m \subseteq \mathsf{I} \times \mathsf{I} \times \mathsf{K}$
Structural violations in slice $m \in \mathsf{M}$	$\mathsf{V}_m = \{d \mid d \in \mathsf{N}_m \wedge d \in \mathsf{D}^{\uparrow}\} \cup \{d \mid d \in \mathsf{X}_m \wedge d \notin \mathsf{D}^{\uparrow}\}$
All structural violations	$\mathsf{V} = \bigcup_{m \in \mathsf{M}} \mathsf{V}_m$

every lifted dependency (in $\mathsf{D}^{\uparrow}$) to easily generate error messages that reference the source code element that leads to a violation. Finally, the set of all violations V is a union over all violations V_m found in all slices $m \in M$.

The sets of not allowed and expected dependencies (N_m and X_m) are derived as depicted in Table 4, which is described next. In essence, the derivation of these sets discriminates between the different types of constraints. For example, the set N_m is a union over five different sets; one for each different type of constraint, i.e., not_allowed, global/local_incoming and global/local_outgoing (the sets are defined in this order in Table 4). In the following we describe the derivation of the sets for global_ − /local_incoming in detail. The sets for global_ − /local_outgoing are symmetric, i.e., the former constrain a target ensemble e_t and the latter a source ensemble e_s.

To simplify the formulas for not allowed dependencies, we use an auxiliary set C_m^+ that represents all ensembles that are **allowed** to use the target of an incoming constraint or the source of an outgoing constraint (w.r.t. the constraints defined in slice m). For incoming constraints C_m^+ allows dependencies in the following cases: (i) A global or local constraint is defined between the source and target ensemble, and (ii) for global constraints all descendants of a source ensemble are also allowed to use the target ensemble. The special treatment of descendants is necessary — though it slightly complicates the checking process — because the descendants' extents overlap with the extents of their ancestors and, hence, share many dependencies. For illustration consider the global constraint in Fig. 1. The ensemble SessionManagement has several descendants, e.g., ConcreteSession or SessionFactory that have their own extents. Some of these descendants have dependencies to HQL and, thus, must be allowed to use HQL. In fact, the global constraint on the parent SessionManagement states that we know some internal source code elements (contained also in descendants) are

Table 4. Derivation of Not Allowed and Expected Ensemble Dependencies

$$\begin{aligned}
\mathsf{C}_m^+ = \ & \{(\text{allowed_incoming}, e_s, e_t, k) \mid e_s, e_t \in \mathsf{L}_m, k \in \mathsf{K} \wedge \\
& \quad (\text{local_incoming}, e_s, e_t, k) \in \mathsf{C}_m \vee \\
& \quad \big(\exists e \in \mathsf{L}_m \wedge (\text{global_incoming}, e, e_t, k) \in \mathsf{C}_m \wedge ((e, e_s) \in \mathsf{H}^* \vee (e_s = e))\big)\} \\
& \cup \{(\text{allowed_outgoing}, e_s, e_t, k) \mid e_s, e_t \in \mathsf{L}_m, k \in \mathsf{K} \wedge \\
& \quad (\text{local_outgoing}, e_s, e_t, k) \in \mathsf{C}_m \vee \\
& \quad \big(\exists e \in \mathsf{L}_m \wedge (\text{global_outgoing}, e_s, e, k) \in \mathsf{C}_m \wedge ((e, e_t) \in \mathsf{H}^* \vee (e_t = e))\big)\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{N}_m = \ & \{(e_s, e_t, k) \mid e_s \in \mathsf{I}, e_t \in \mathsf{I}, k \in \mathsf{K} \wedge \exists(\text{not_allowed}, e_s, e_t, k) \in \mathsf{C}_m\} \\
& \cup \{(e_s, e_t, k) \mid e_s \in \mathsf{I}, e_t \in \mathsf{L}_m, k \in \mathsf{K} \wedge \\
& \quad \exists e \in \mathsf{L}_m.\, (\text{global_incoming}, e, e_t, k) \in \mathsf{C}_m \wedge (\text{allowed_incoming}, e_s, e_t, k) \notin \mathsf{C}_m^+\} \\
& \cup \{(e_s, e_t, k) \mid e_s \in \mathsf{L}_m, e_t \in \mathsf{L}_m, k \in \mathsf{K} \wedge \\
& \quad \exists e \in \mathsf{L}_m.\, (\text{local_incoming}, e, e_t, k) \in \mathsf{C}_m \wedge (\text{allowed_incoming}, e_s, e_t, k) \notin \mathsf{C}_m^+\} \\
& \cup \{(e_s, e_t, k) \mid e_s \in \mathsf{L}_m, e_t \in \mathsf{I}, k \in \mathsf{K} \wedge \\
& \quad \exists e \in \mathsf{L}_m.\, (\text{global_outgoing}, e_s, e, k) \in \mathsf{C}_m \wedge (\text{allowed_outgoing}, e_s, e_t, k) \notin \mathsf{C}_m^+\} \\
& \cup \{(e_s, e_t, k) \mid e_s \in \mathsf{L}_m, e_t \in \mathsf{L}_m, k \in \mathsf{K} \wedge \\
& \quad \exists e \in \mathsf{L}_m.\, (\text{local_outgoing}, e_s, e, k) \in \mathsf{C}_m \wedge (\text{allowed_outgoing}, e_s, e_t, k) \notin \mathsf{C}_m^+\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{X}_m = \ & \{(e_s, e_t, k) \mid e_s, e_t \in \mathsf{I}, k \in \mathsf{K} \wedge \exists(\text{expected}, e_s, e_t, k) \in \mathsf{C}_m \vee \\
& \quad \exists(\text{global_incoming}, e_s, e_t, k) \in \mathsf{C}_m \vee \exists(\text{local_incoming}, e_s, e_t, k) \in \mathsf{C}_m \vee \\
& \quad \exists(\text{global_outgoing}, e_s, e_t, k) \in \mathsf{C}_m \vee \exists(\text{local_outgoing}, e_s, e_t, k) \in \mathsf{C}_m\}
\end{aligned}$$

allowed to use HQL. Yet, at this level of abstraction, we do not model which particular code elements are allowed to have the dependencies. This treatment of descendants ensures that hierarchies have an identical semantics for a "flat" ensemble with a larger extent and an ensemble subdivided by descendants. In other words, if we would model SessionManagement with an extended query (that includes the code elements of all descendants), but without descendants, we do specify the same constraint. Note that the distinction is necessary only for global constraints, because these constraints quantify over all ensembles and, hence, also over the descendants. Note further that no special treatment is necessary for the target of the incoming constraint (e.g., HQL), since the target contains all code elements of all descendants. I.e., we implicitly constraint the usage of any code element contained in any descendant. Allowed dependencies for outgoing constraints are reflected in C_m^+ accordingly with source and target ensembles reversed.

For the derivation of **not allowed** dependencies (N_m) we distinguish five cases. First, if a not_allowed constraint exists from some ensemble e_s to another ensemble e_t, then corresponding dependencies are not allowed. Second, if a global_incoming constraint exists to e_t, all other source ensembles of the project are not allowed to use e_t, unless they are explicitly allowed, which means a respective entry can be found in the set C_m^+. Third, constraints of type local_incoming

have a similar semantics to those for global_incoming, with the difference that the target of a disallowed dependency must be in the set L_m, i.e., the set of ensembles used in the slice m, and not in I, the set of all ensembles. Note that for slices using mixed declarations of global and local constraints no special treatment is required. In this case, the set of ensembles that are not allowed to use the target (e_t) due to global_incoming constraints is a superset of the set deduced due to local_incoming; since $\mathsf{I} \supseteq \mathsf{L}_m$. The fourth and fifth case treat global and local outgoing constraints accordingly (again with the source and target ensembles reversed).

For the derivation of **expected** dependencies (X_m) we also distinguish five cases. First, constraints of type expected specify that there exists at least one lifted dependency from the ensemble e_s to the ensemble e_t. For the other four cases — global_ − /local_incoming and global_ − /local_outgoing — we also check that at least one lifted dependency from the ensemble e_s to the ensemble e_t exists. The rationale is that these cases list ensembles that are specifically allowed to use another ensemble and that it is dubious that they are included in the list if there are no dependencies in the source code. The semantics for expected dependencies again treats hierarchies such that the behavior is identical whether an ensemble has child ensembles or not. In contrast to the not allowed dependencies we require no special treatment of descendants in the checking process.

3.4 Tool Support

Vespucci supports the following tasks: (i) defining the mapping between ensembles and code, as well as analyzing the mapping for consistency when the software evolves, (ii) modeling of dependency constraints, (iii) checking the actual implementation against the modeled constraints and reporting violations. The modeling of constraints (ii) is supported by a graphical editor for the definition of slices as shown in section 3.2 and is not further discussed. The support for the other tasks is described next.

Mappings between the source code and the ensembles are defined in the query language introduced in Section 3.1. In the following, we present the use of the query language in more detail and in particular how Vespucci supports the adaptation of the queries during the evolution of Hibernate's structure. The latter is especially important to ensure a meaningful, comprehensive specification that takes all source code elemants into consideration.

The evolution of the `Cache` ensemble is a good example candidate to present Vespucci's support. This ensemble changed four times during our study of the evolution of Hibernate. Initially, the `Cache` ensemble contained all classes in the `cirrus.hibernate.cache` package. The initial mapping was adapted in version 1.2.3 to the query depicted in Fig. 8. In this version the class `CacheEntry` was created, but it was a member of the general `impl` package though conceptually belonging to the caching concern. Thus, the class was not captured by the original query. To identify classes, such as `CacheEntry`, Vespucci has a specialized view (cf. Fig. 7) that shows all code elements that currently do not belong to any ensemble.

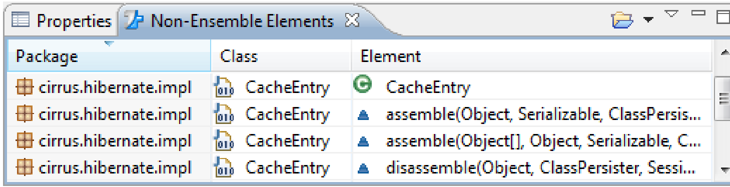


Fig. 7. View of Elements that do not belong to any Ensemble

After identifying that `CacheEntry` belongs to the Cache ensemble we extended the original query to include the class. The query can either be directly edited in the respective editor or changed via drag and drop of code elements onto ensembles. Using the drag and drop functionality extends (or creates) a query such that all respective code elements will be in the extent of the ensemble. For illustration the extended query is shown in Fig. 8. Keywords of the query language are highlighted. In Fig. 8 the `package` sub-query was already in the initial model for Hibernate. The `class_with_members` query for the `CacheEntry` class was created using drag and drop.

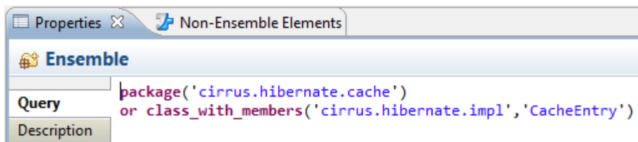


Fig. 8. Query Editor showing the Adapted Query for the Cache Ensemble (in Hibernate 1.2.3)

The code belonging to the Cache ensemble was refactored by the Hibernate developers in a later version (3.0). In that version, cache entries with different behaviors were introduced as separate classes and all of those were put into the package `hibernate.cache.entries`. These code changes led to warnings that the respective packages contain unmapped elements and that the current sub-query for `CacheEntry` does not select any elements. Thus, we immediately saw that the code was changed w.r.t. our previous assumptions about the implementation of the caching functionality.

In summary, both mechanisms for detecting changes, i.e., support for identifying unmapped elements and queries that select no elements, are important to help the architect to maintain the mapping between source code elements and ensembles. The view on empty queries is in particular useful for fine-grained ensembles in lower levels of the hierarchy. These ensembles typically consisted of only a few classes in our study and naming patterns were not always sufficient to formulate a query encompassing all classes. Hence, some classes were enumerated by fully qualified names and when these classes were refactored the view on empty queries provided the information which queries needs to be maintained. Note that making Vespucci aware of refactorings would also help to alleviate

this situation, i.e., if Vespucci would be refactoring aware, the queries/ensembles that are affected could be identified and presented to the user. In some cases it would be possible to automatically adapt a query when a class' name is refactored. Yet, IDE refactoring tools can still be circumvented by developers. Hence, support for checking the consistency of the mapping is indispensable for a tool for structural dependency checking.

The implementation of Vespucci's dependency checker (iii) is integrated into the Eclipse IDE and checking is done as part of Eclipse's incremental build process. In general, checking is done against the constraints defined in all slices and is always performed for an entire Eclipse project. This includes the compiled source code of the project as well as the used libraries. Additionally, Vespucci supports checking of individual slices to facilitate the modeling of the dependency constraints. When checking is enabled, any violation of a constraint in the checked slices is reported in Eclipse's "Problems View." After every change, the code is re-analyzed and the "Problems View" is updated. The checking is done incrementally ([17]) and is efficient enough for (at least) mid-sized projects such as Hibernate.

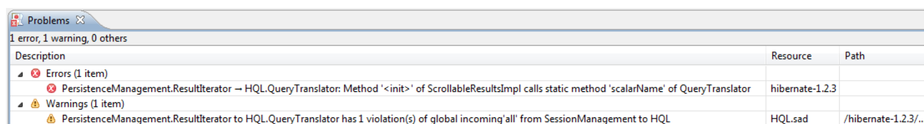


Fig. 9. A Violation of the Dependency Rule for Hibernate Query Language (Reported in Hibernate Version 1.2.3)

Figure 9 depicts an example of a violation that was reported when checking our model of Hibernate against the source code of the version 1.2.3. The actual violation in the code is indicated via an error marker that can be used to navigate to the respective line in the source code. The error message lists the source and target ensemble , the code elements, e.g., methods with names and containing class, as well the dependency that violates a constraint, e.g., method A "calls" method B. In addition to marking the code elements, a summary of all violations per source/target ensemble pair is shown as a warning. The summary facilitates a quick overview in case that many code elements in the source and target ensemble have violations. Furthermore, the summaries are marked as warnings of the actual slices in which the violation was detected and can be used to navigate to the particular slice.

Vespucci furthermore supports the explicit documentation of acknowledged violations. This supports architecture reviews as well as gradually refactoring of an existing code base toward the intended architecture. In the latter scenario, documented violations are used to guide the refactoring efforts of developers. Note that the typical workflow of Vespucci is to enable checking against the architecture while developing the code. Hence, dependencies that violate the intended design are ideally removed immediately during development. Yet, the above scenarios for reviews and gradual refactoring are supported as well. Figure 10 illustrates how the slice

from Fig. 1 was modified to document the violation shown in Fig. 9. The violation is marked as a dependency with a warning sign. If the intended refactoring is already clear, it can be outlined in a comment in the slice. Documented violations are also removed from the general error reporting and can be reviewed in a different view.

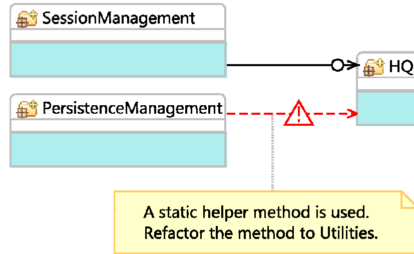


Fig. 10. Documenting a Violation in a Slice for Subsequent Refactoring

3.5 On Modeling Methodology

Breaking down the architecture of a system into multiple slices comes with a new set of modeling decisions. Figure 11 schematically shows four principal ways to model the architecture of a hypothetical system consisting of four ensembles (boxes labeled 1 to 4) with Vespucci. In (A), all constraints are modeled in a single model, i.e., Vespucci is used with a single slice (dashed box around the ensembles) In (B), the model makes use of hierarchical structuring – specifically, ensembles 1 and 2 are nested into an ensemble 1&2. In (C), the model makes use of slicing; specifically, per ensemble one slice is defined that models all structural dependency constraint w.r.t. that ensemble. This includes all incoming and all outgoing dependencies. However, slicing at other granularity levels is conceivable (see below). In (D), the model makes use of both slices and hierarchies, which is the expected typical usage of Vespucci.

In general, the structural dependency model of a system in Vespucci consists of an arbitrary number of slices. It is a matter of modeling decisions – taken by the architect – in how many slices she breaks down the overall architectural specification. As part of this process, a trade-off is to be made between (i) creating (large(r)) slices that capture several architectural rules related to multiple ensembles that conceptually belong together and (ii) creating one slice per ensemble that just captures the architectural rules related to that ensemble. In the former case cohesiveness is fostered while in the latter case (local) comprehensibility of the architecture and evolvability of the specification is fostered.

In the Hibernate case study, as a rule of thumb, each high-level slice focused on design decisions concerning one ensemble. For instance, the slices in Fig. 1 and Fig. 3 focus on specific design decisions related exclusively to allowed *incoming dependencies* to HQL, respectively allowed *outgoing dependencies* of ConnectionProvider. Internal dependencies for ensembles with nested sub-ensembles were in general related to a small set of ensembles and hence captured in a single slice, as e.g., in Fig. 4, where the internals of SQLDialects were captured.

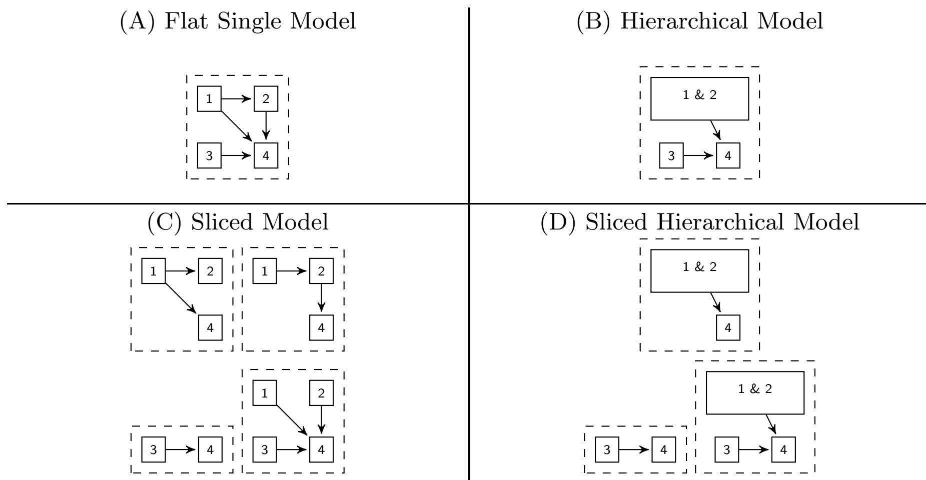


Fig. 11. Alternative Architectural Models of Dependencies

The one-slice-per-high-level-ensemble strategy for breaking down specifications is just a first approximation. For reasons of better managing complexity and evolvability as well as understandability, it may make sense to chose more fine-grained or coarse-grained strategies. One such strategy is to split the specification of incoming and outgoing dependencies of an ensemble, if those are too complex or evolve in different ways. On the other hand, slices of related ensembles may be merged, when their separated specifications are too simple to justify separate slices or hard to understand in isolation.

One may criticize that a specification becomes complex with an increasing number of slices. However, a single specification that controls the dependencies to the same degree is no less complex and includes all information that are captured in the slices. For example, if internal dependencies are controlled, they need to be specified and maintained in a single specification as well. The focus here is to make a case for enabling the architects to break down specifications of structural dependencies in several modules that are more manageable w.r.t. scalability and evolvability and can be reasoned over in isolation. Hence, slices also facilitate distribution of work, such that large architectures can be maintained by a team rather than a single architect.

Per ensemble slicing of the dependency model may also impair understandability of dependencies pertaining to several modules. A view of the dependencies for multiple ensembles (in contrast to their individual constraints) can be advantageous for the exploration of the architecture, e.g., if one wants to follow transitive dependencies such as the path of communication from ensemble A to B. Note that if such a path is relevant to the architect, it can also be encoded as a slice. A second scenario for global comprehension is to find all slices in which an ensemble participates. This can be supported by a simple analysis over the defined slices.

All the above said, systematically deriving guidelines for structuring architectural decisions into slices and distributing the work is a matter of performing comprehensive studies and is out of the scope of this paper.

3.6 Scalability and Evolvability

Vespucci enables architects to reason about architectural decisions concerning structural dependencies of a set of ensembles in isolation, while treating the rest of the system as a black-box, and to do so in a top-down manner. This is due to (1) Vespucci's support for breaking down the specification into slices, (2) mechanisms for expressing structural rules via a constraint system, (3) a scoping mechanism that enables to quantify locally or globally over the set of affected ensembles, and (4) Vespucci's support for enabling the hierarchical organization of specifications. The latter is a traditional mechanism to govern complexity [18] and will for this reason not be further considered in the following discussion.

Support for modular reasoning. Slices enable the architect to focus on constraints that concern individual ensembles or a set of strongly related ensembles. This makes it possible to isolate a small set of related architectural decisions from the rest for the purpose of modularly reasoning about them, while treating the rest as a black-box.

This fosters scalability by reducing the number of ensembles and constraints that need to be considered at once: Each slice in Fig. 11 (C) contains less ensembles and constraints than the model in Fig. 11 (A). One may argue that slicing actually increases the overall number of elements (ensembles/constraints) — since some of them are mentioned in multiple slices. However, as they represent the same abstractions in all slices, the overall number of elements that need to be understood remains the same as in the model A.

Consider for illustration the slice depicted in Fig. 2. It expresses that only SessionManagement and PersistenceManagement may use SQLDialects with the minimum amount of explicitly mentioned ensembles and constraints. No rules governing dependencies between SessionManagement and PersistenceManagement, respectively between those and other ensembles, are specified. The slice in Fig. 2 models architectural constraints from the perspective of SQLDialects. Dependencies between SessionManagement and PersistenceManagement or between those and other ensembles are irrelevant from this perspective and are, thus, left unspecified. Further, we do not explicitly enumerate all ensembles that are not allowed to depend on SQLDialects.

Vespucci's constraint system for modeling dependencies and the way checking for architecture compliance operates (see previous section) is key to the conciseness of specifications. Slices are checked in isolation. The constraint system interprets the lack of a constraint in a slice as "don't care" in the sense that the presence or absence of code dependencies is ignored. E.g., potential dependencies between SessionManagement and PersistenceManagement are ignored when checking compliance with the rules defined by the slice shown in Fig. 2. They may well be the subject of specification in other slices to be reasoned on separately.

The role played in this respect by our distinction of incoming and outgoing constraints needs to be highlighted here. It is the use of the incoming constraints in Fig. 2 that enables us to talk about constraints from the perspective of SQL-Dialects – excluding from consideration any further dependencies in which, e.g., SessionManagement may engage. Incoming/outgoing constraints are "unilateral" – they belong to one ensemble. Without this distinction, we would be left with "bilateral" constraints; mentioning one such constraint that affects SessionManagement would require to mention all other constraints affecting SessionManagement; hence, making it impossible to slice specifications.

The ability to abstract over any dependencies that are not explicitly constrained comes in also very handy when handling ensembles that are expected to be ubiquitously used, e.g., Hibernate's Utilities ensemble. Such ensembles would typically contribute a significant amount of complexity to architectural specifications, if the specification approach requires to explicitly mention allowed dependencies. By using a constraint system this complexity can be avoided. The specification would make no mention of dependencies to Utilities, in order to leave it unconstrained.

The ability to state a constraint that affects arbitrary many ensembles without having to enumerate those explicitly is due to the ability to make global statements. Ensembles that are not explicitly mentioned in a slice are reasoned over by global constraints, e.g., the slice shown in Fig. 2 implicitly states that all other ensembles mentioned in Fig. 1, 2, 3, and many more, are not allowed to use SQL Dialects. This specification is much smaller compared to enumerating this fact for all other ensembles constituting the rest of Hibernate. The latter would be necessary, if Vespucci only had allowed and not-allowed constraints and no distinction between local and global scopes.

Support for evolution. Due to slicing, architectural models also become easier to extend. First, slices remain stable in case of extensions that do not affect their ensembles/constraints. Second, affected slices are easier to identify. Finally, existing global constraints automatically apply to new ensembles.

Consider for illustration the following scenario that occurred during the evolution of Hibernate from version 1.0 to version 1.2.3. In this step, a new ensemble — called Metadata — to represent Hibernate's new support for metadata was introduced. This change was accommodated mostly incrementally. First, the specification as a whole was extended incrementally by introducing a new slice, referring to the ensembles that Metadata is allowed to use and be used from. Second, the set of existing slices that eventually required revision was restricted to those modeling the dependencies of ensembles referred to in the new Metadata slice. For example, the slice that defined constraints for DataTypes was refined to enable the usage by Metadata. Slices that modeled unrelated architectural decisions, e.g., those governing dependencies of ConnectionProvider (cf. Fig. 3), did not require any reviewing. Yet, previously stated global constraints carry over to the new ensemble, ensuring for example that it does not unintentionally use SQLDialects (slice in Fig. 2); the usage of non-constrained ensembles, e.g., Utilities, is also granted automatically.

The way the mapping between ensembles and source code is modeled has an effect on the stability of the model in face of evolution of the system. Here we hit a variant of the well-known "fragile pointcut problem". One way to mitigate this problem is by using stable abstractions in the source code queries. However, this is not always feasible; in our case study we had to adapt queries as the system evolved. Here, the tool support provided by Vespucci offered some help to identify changes in the source code by: a) showing elements that do not belong to an ensemble, b) showing (sub-)queries with empty results, and c) specifying that a list of ensembles should be non-overlapping (i.e., to prevent accidental matches). Even so we are aware that better source code query technology and tool support for it is needed; in this paper, we focus on the modularity mechanisms on top of the query language.

4 Evaluation

In this section, we evaluate quantitatively the effectiveness of Vespucci's mechanisms to modularize the specification of a software's intended structure. This evaluation is performed from two complementary perspectives: (a) reduction of complexity, which is measured as the number of ensembles and constraints, and (b) facilitating architecture maintainability during system evolution. As a basis we use the re-engineered architecture of Hibernate (c.f. Sec. 2), which allows us to study an architecture of a size that is representative for mid- to large-scale projects. We also give a critical discussion of the broader applicability of our results and of threats to the validity of our study at the end of the section.

The goal of our evaluation is to assess the modularization mechanisms of Vespucci and not the accuracy of architectural violation control. Therefore, even though Vespucci is targeted at continuous architecture conformance checking, the identification of architecture violations is not the purpose of our quantitative evaluation. Nevertheless, it is important to highlight that in terms of enforcing conformance, Vespucci is at least as capable as related approaches [6–9, 19–21].

4.1 Scalability

We first analyze the reduction in complexity when reasoning about an architecture specification. This analysis was performed by comparing the architecture of Hibernate 1.0 modeled in the four principal ways schematically depicted in Fig. 11 and outlined in the previous section. The model with both slices and hierarchies (Fig. 11, D) was the primary model produced during our study of Hibernate. The other three models were produced to measure the complexity reduction for the different mechanisms (hierarchies, slices, combination of both).

Scalability with regard to the number of ensembles We first compare different mechanisms w.r.t. the number of ensembles referenced by isolated dependency rules. The baseline is a single monolithic specification with a total of 79 ensembles, modeled by following Fig. 11 (A). The other three models Fig. 11

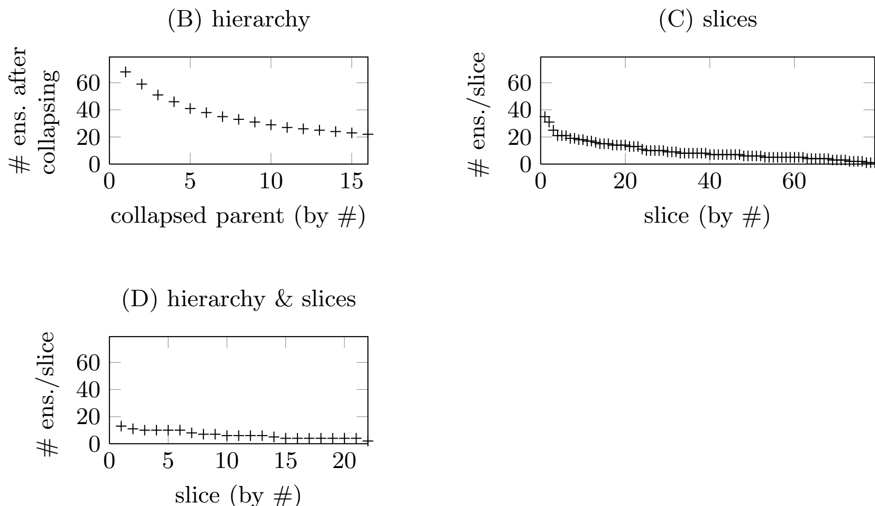


Fig. 12. Comparison of ensemble reduction w.r.t. hierarchies and architectural slices (Hibernate 1.0)

(B-D) are quantified in the diagrams in Fig. 12. The y-axis of all three diagrams denominates the number of ensembles referenced per architectural model.

The diagram on the top left shows reduction in complexity for hierarchical structuring only. The model is a single specification, but high-level ensembles may be collapsed to reduce the overall number of ensembles to consider at once. The x-axis denominates the number of collapsed ensembles ordered by the number of their sub-ensembles. The values on the y-axis show how many ensembles are referenced after collapsing an enclosing ensemble, i.e., the enclosing ensemble is referenced instead of all its children. The values are accumulated, since multiple ensembles can be collapsed together. For example, in a model with the top five most complex high-level ensembles collapsed, the architect has to consider 41 ensembles at once. When collapsing all ensembles in the hierarchy, we are left with 22 top level ensembles, hence hierarchical structuring reduces the number of ensembles to approx. 27% of the total (22 of 79).

The diagram in the top right of Fig. 12 shows the number of ensembles per slice when using only slices (no hierarchies). The x-axis denominates the modeled slices in the decreasing complexity order (decreasing number of referenced ensembles). Almost all slices refer to less than 27% of the ensembles (12% on average). The exemption are the three first slices that capture rules for the following building blocks (of central importance) (i) persisting classes, (ii) persisting collections, and (iii) the interface to Hibernate's internal data types. The combination of both mechanisms (diagram on the bottom left of Fig. 12), yields a much smaller number of slices (x-axis), since it focuses on the top-level building blocks. In addition, the combined approach features slightly smaller slices; on average each slice references only 9% of the total number of ensembles.

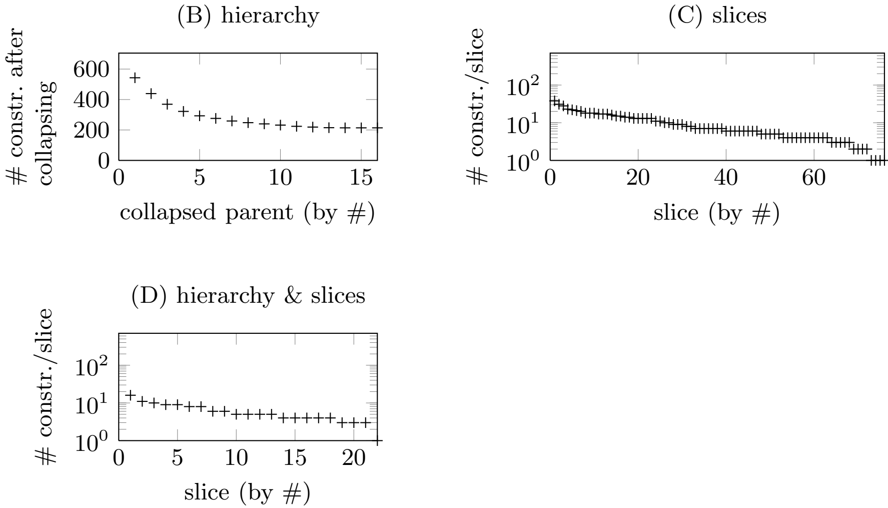


Fig. 13. Comparison of constraint reduction w.r.t. hierarchies and architectural slices (Hibernate 1.0)

Scalability with regard to the number of constraints. In the following, we compare how much each mechanism reduces the number of constraints used in isolated dependency rules. The comparison is similar to the comparison regarding the number of ensembles and the numbers are shown in Fig. 13. The x-axis is organized in the same manner as in Fig. 12. The y-axis denominates the number of constraints that are referenced in each architectural model.

The y-axis for hierarchical structuring (top left diagram in Fig. 13) shows the total number of constraints after collapsing an enclosing ensemble. The number includes (i) constraints that are abstracted away, since they are internal to the enclosing ensemble (cf. Fig. 11 B; 1&2) and (ii) constraints that are abstracted away, since several constraints at the low level are subsumed by a single constraint at the high level (cf. Fig. 11 B; 1&2 to 4). Both internal and external constraints contribute approx. half of the reduction in constraints (external slightly outweighs internal). As in the evaluation for ensembles, the y-values for the hierarchical composition (B) are accumulated, since we can use several hierarchical groupings together. For the architectural models using slices (C,D) the number of constraints is simply the number of constraints modeled in one slice.

The baseline (A) consists of 705 constraints in a single specification. If we consider the hierarchical model and collapse all enclosing ensembles, approx. 2/3 of the constraints are removed (down to 214, last value in the top-left diagram in Fig. 13)). In comparison, slices (diagram in the top right of Fig. 13) show less than 5% of the total number of constraints and 1,3% on average (9 of 705) per slice. The combination of slices and hierarchical structuring (bottom left diagram in Fig. 13) features slightly smaller slices; on average 0.9% (6.5 constraints) of the total of 705 constraints modeled.

Scalability with regard to the number of slices. To control the architecture of Hibernate we have modeled top-level slices comparable to Fig. 11 (D) and slices for the internal constraints of the 16 ensembles that are further structured; totaling to 35 slices. Thus, the overall number of slices is smaller than the overall number of ensembles (79) and remains manageable. Note that in these models we do not use the total of the 705 constraints. First, three ensembles at the top level (SessionManagement, PersistenceManagement, and UserAPI) have no slice (and no constraints), for the reason of being used by and/or using almost all other ensembles. Note that this applies only to the mentioned top-level ensembles. In Hibernate many ensembles require session functionality and, hence, use SessionManagement. PersistenceManagement on the other hand uses data from many other ensembles to fulfill it's functionality. However, the high coupling points to deficiencies in the modularization of the software system and it is conceivable to reduce the coupling. Second, the modeled top-level constraints subsume several constraints on the internal ensembles. We found the control provided by the top-level constraints mostly sufficient during the evolution of Hibernate. Hence, we modeled detailed constraints only in few cases to further our understanding of the dependencies between selected ensembles. This was done, for example, to more carefully control that only the sub-ensembles that represent SessionManagement's interfaces are used by other ensembles.

Summary. In this study the hierarchical structuring included 22 ensembles and 215 constraints (both approx. 1/3 of the total). Slices are much smaller; we have to collapse the first seven enclosing ensembles of the hierarchy to reduce the number of ensembles to 35, the number referenced in the most complex slice (persisting classes). Collapsing all ensembles still references 5 times more constraints than the number referenced in the slice for persisting classes. Hence, the modeling approach based on slices scales much better by reducing each slice to 9.5 ensembles and 9 constraints on average. The combination of both mechanisms produces the best results by reducing each slice to 7.1 ensembles and 6.5 constraints on average, which means that a typical slice in the Hibernate model had about 7 ensembles and 6 to 7 constraints. Thus especially the number of constraints that need to be reasoned over at once remains manageable and includes on average only 3% of the constraints of the model using hierarchical structuring, with a maximum of 16 constraints, or 7% of the constraints in the single hierarchical model.

4.2 Evolvability

To evaluate the effectiveness of Vespucci in supporting architecture evolution we have compared a single model with hierarchies (Fig. 11, B) to slices with hierarchies (Fig. 11, D). The results are summarized in Table 5. The first three Columns show the analyzed version, its release year, and the number of LoC as an estimate for the size. Columns four and five characterize the architecture evolution in terms of ensembles and their queries. Overall, the number of ensembles has doubled. Column six shows the total number of slices in each version. We

followed the methodology of one slice per ensemble – hence, the number of slices roughly follows the number of ensembles, with the exception of those ensembles that were not constrained (c.f. Sec. 4.1). Column seven shows that on average 33% of all slices (1/3 of the architecture specification) remained stable w.r.t. the previous version. The least stable revisions were the first and the last one. In the first revision, Hibernate was close to its inception phase, hence requiring more adaptations to its features. The last revision was the most extensive in terms of the timespan covered. The last three columns compare the complexity involved in performing the required updates of the architecture specifications. Columns eight and nine show the average, resp. maximal number of ensembles per slice, whose dependencies were updated, in the approach using slicing. The last column shows how many dependencies were updated in the single hierarchical model. On average only 4% to 6% of the number of dependencies updated in the single model were reviewed per slice (the maximum ranging between 7% and 15%). This reduction in complexity of the updates per slice is comparable with the reduction of the number of constraints between (B) and (D) in Fig. 13.

The numbers indicate that the maintenance of individual slices is much easier than the evolution of the single architecture model and confirm what is qualitatively discussed in the previous section.

Table 5. Analysis of the Evolution of Hibernate's Architecture

							Dependencies reviewed using:		
Version	Release Year	LoC	# Ensembles (Top-Level)	Added/Removed Ensembles	# Slices (Top-Level)	Stable Slices	Slices (Avg.)	Slices (Max.)	Single Model
1.0	2002	14703	22	n/a	19	n/a	n/a	n/a	n/a
1.2.3	2003	27020	26	+5 / -1	23	4 (21%)	2.4	6	61
2.0	2003	22876	28	+5 / -3	25	12 (52%)	1.9	6	40
2.1.6	2004	44404	30	+2 / -0	27	9 (36%)	2.6	6	38
3.0	2005	79248	36	+9 / -3	33	8 (30%)	4.5	8	118
3.6.6	2011	106133	39	+3 / -0	36	9 (27%)	5.0	11	87

4.3 Threats to Validity

We identify two threats to the *construct validity* of our study. First, the reverse engineering of Hibernate's architecture was primarily performed by this paper's authors, i.e., not by the original Hibernate developers. Hence, the resulting architecture design may not accurately reflect Hibernate's real/intended architecture, which may lead to inconsistencies in the results. To mitigate this threat, the architectural model was created by three people — one student, one

PhD candidate and one post-doctoral researcher — that together have many years of experience on object-relational mapping frameworks. Further, we extensively studied the available documentation to make sure that the model is true to Hibernate's architecture. Yet, it is likely that a different group would reverse engineer a different architectural model. But, it is unlikely that the architecture would be such different that our evaluation would become invalid. A second threat to construct validity is that other architects may modularize the architecture specification differently, resulting in a different number and scope of slices. However, the approach that we followed — roughly creating one slice per top-level ensemble — has proven to be useful and can at least be considered as one reasonable approach.

Threats to *conclusion validity* in our study could be related to the number of ensembles and architectural constraints involved in our analysis. We tried to mitigate this threat by considering an architectural model of a significant complexity. Our analysis concerned an architectural model that involved 79 ensembles, more than 700 architectural constraints and 35 architectural slices for Hibernate 1.0.

The main issue that threatens the *external validity* of our study is that it involved a single software system. To mitigate this threat we have used a well-known medium-size framework, which has been designed by taking into consideration guidelines and good practices. These characteristics allow us to analyze the benefits of Vespucci when modeling architecture designs of well-modularized software systems. In addition, we have discussed the properties of Hibernate's architecture that influence the results and compared them to other studies. However, we are aware that more studies involving other systems should be performed in the future. All our findings should be further tested in repetitions or more controlled replications of our study.

5 Related Work

Closely related to Vespucci are approaches that support checking the conformance between code and architectural constraints on static dependencies [6–9, 19–21]. The key difference is that none of the above approaches (nor other related work) offers the ability to modularize the architecture description into arbitrary many slices. They rather require a self-contained monolithic specification of the architecture, which does not support the kind of black-box reasoning enabled by slices (cf. Sec. 3.6). In the following, we discuss the above approaches separately; a summary of their support for the features elaborated in Sec. 3 is presented in Table 6.

Reflexion Models (RM) [7] pioneered the idea of encoding the architecture via a declarative mapping to the source code. RM is an analytical approach that uses the modeled system architecture to generate deviations between source code and planned architecture, which is reviewed by the architect. The RM approach is not a constraint system, but rather requires the specification of the complete set of valid dependencies. Omission of dependencies is interpreted as

"no dependency is allowed". Other approaches extend RM by (i) incorporating hierarchical organization [19], (ii) visual integration into the Eclipse IDE [12] and (iii) extending the process to continuously enforce compliance of structural dependencies between a planned architecture and the source code [13].

Sangal et al. [8] discuss the scalability issue of architecture descriptions and propose a hierarchical visualization method called design structure matrices (DSMs), which originates from the analysis of manufacturing processes. The key advantage is the notation (matrices) that facilitates identification of architectural layers via a predominance of dependencies in the lower triangular half of the matrix. DSM features a very verbose constraint system. For example, exemptions on lower level ensembles are encoded by the order in which rules are declared, e.g., by first allowing PersistenceManagement to use SQLDialects and then disallowing the use of ConcreteDialects. While effective, this approach requires a carefully crafted sequences of constraints. The work on DSM culminated in a commercial tool called Lattix, which has since evolved and gained a classification system for dependencies similar to our dependency kinds.

Table 6. Comparison with the State of the Art

	Reflexion Models (RM) [7]	Hierarchical RM [19]	DSM [8]	LogEn/VisEn [6]	DCL [9]	Vespucci
Architectural slices	-	-	-	-	-	✓
Constraint system[1]	-	-	+	+++/-[2]	++	+++
Hierarchies	-	✓	✓	✓	-	✓
Dependency Kinds	-	-	(✓)	-	✓	✓

[1] - (non existent) to +++ (very expressive)
[2] LogEn is very expressive; VisEn does not offer a constraint system

In previous work [6] we proposed an approach to continuous structural dependency checking; integrated into an incremental build process. As in Vespucci, we referred to conceptual building blocks as ensembles. However, the specification of architectural constraints has been completely revised for Vespucci. Previously we have defined LogEn; a first order logic DSL, that integrated query language and constraint specification. However, the meaning of a violation, i.e., a constraint, is defined by the end-user, which is complex in first order logic. Hence, we provided a visual notation (VisEn), which is less complex, but focuses on documenting the architecture and hence is not a constraint system, but requires explicit modeling of all dependencies. The focus of this work was on the efficient incrementalization of the checking process, hence slicing architecture specifications into manageable modular units was not supported.

Terra et al. [9] propose a dependency constraint language (DCL) that facilitates constructive checking of constraints on dependencies; discrimination of dependencies by kind is also supported. DCL offers a textual DSL for specifying constraints. DCL's constraint system is closest to Vespucci's, and can express the not-allowed, expected and incoming constraints. Yet, it lacks outgoing constraints and a scoping mechanism such as global/local constraints, which goes hand in hand with the lack of support for slicing specifications into modular units. The language supports no inherent hierarchical structure in the architecture.

A number of commercial tools have been documented (c.f. [21]) for checking dependency among modules and classes using implementation artifacts, e.g., Hello2Morrow Sotograph [22]. However, the scope of these tools is limited; they are only able to expose violations of "certain" architectural constraints such as inter-module communication rules in a layered architecture. That is, they do not provide means for expressing system constraints.

In [20] the authors propose a technique for documenting a system's architecture in source code (based on annotations) and checking conformance of code with the intended architecture. The representation of the actual architecture in the source code is hierarchical, however, they do not support slicing of specifications in modular units and the modular architectural reasoning related to it.

Languages specialized on software constraints like SCL [23], LePUS3 [24], Intensional Views [25], PDL [26] and Semmle .QL [27] can be used to check detailed design rules e.g., related to design patterns [28]. However, they are not expressive enough for formulating architectural constraints in a way that allows to abstract over irrelevant constraints, when reasoning about a part of the architecture in isolation.

In [29] authors introduce a technique to identify modules in a program called concept analysis. A concept refers to a set of objects that deal with the same information. The authors observed that, in certain cases, there is an overlap among concept partitions. The notion of slice in Vespucci could be considered as conceptually close to the notion of concept overlapping since Vespucci supports the grouping of ensembles that are ruled by the same design decisions. Other than that slices and concepts are different in the way they are defined and used. Concepts emerge while slices are explicitly modeled. Moreover, use case slices [30] are also related to our notion of slices, but focus on the modularization of the scattered and tangled implementation of use cases.

In [31] the authors discuss foundations and tool support for software architecture evolution by means of evolution styles. Basically, an evolution style is a common pattern how software architectures evolve. This case study complements our work by helping to identify evolution styles w.r.t. a software's structural architecture. The evolvability of a software that is developed in a commercial context is also discussed by Breivold et al. [32]. They propose a model that — based on a software's architecture — evaluates the evolvability of the software. Based on our experience, the model also applies to open-source software, such as Hibernate. Aoyama [33] presents several metrics to analyze software architecture evolution. He made the general observation that discontinuous evolution emerges between certain periods

of successive continuous evolution. Our case-study confirms this observation. We observed that some parts of Hibernate evolved continuously, while in other parts the evolution was disruptive. Using our modular architecture conformance checking approach architects can focus on continuous and disruptive slices individually.

6 Summary and Future Work

In this paper, we proposed and evaluated Vespucci, an approach to modular architectural modeling and conformance checking. The key distinguishing feature of Vespucci is that it enables to break down specification and checking into an arbitrary number of models, called architectural slices, each focussing on rules that govern the structural dependencies of subsets of architectural building blocks, while treating the rest of the architecture as a black box. Vespucci features an expressive constraint system to express architectural rules and also supports hierarchical structuring of architectural building blocks.

To evaluate our approach, we conducted an extensive study of the Hibernate framework, which we used as a foundation for a qualitative evaluation, highlighting the impact of Vespucci's mechanisms on managing architectural scalability and evolvability. We also quantified the degree to which Vespucci can (a) reduce the number of ensembles and constraints that need to be considered at once, and (b) facilitates architecture maintainability during system evolution. For this purpose, we played back the evolution of Hibernate's structure. The results confirm that Vespucci's is indeed effective in managing complexity and evolution of large-scale architecture specifications. However, given that we have only done one extensive case study so far, we need to carry out further case studies before final conclusions on the scalability of the approach can be made.

In future work, we will explore how IDE support can help to "virtually merge" slices into a virtual global architectural model and to automatically create "on-demand" slices to help architects to plan a software's evolution. Obviously, further empirical studies are needed to better understand the benefits and limitations of Vespucci. New studies need to be designed to asses the impact of the approach on architect's productivity and on software quality. In this respect it would be interesting to study the effect of modularization w.r.t. enlarged control, i.e., the modularization allows to efficiently maintain an architecture containing more ensembles, which provides tighter control over the source code.

References

1. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River (1996)
2. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. IEEE Trans. Softw. Eng. 27(1) (2001)
3. Godfrey, M.W., Lee, E.H.S.: Secrets from the monster: Extracting mozilla's software architecture. In: COSET (2000)

4. MacCormack, A., Rusnak, J., Baldwin, C.Y.: Exploring the structure of complex software designs: An empirical study of open source and proprietary code. Manage. Sci. 52 (2006)
5. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4) (1992)
6. Eichberg, M., Kloppenburg, S., Klose, K., Mezini, M.: Defining and continuous checking of structural program dependencies. In: ICSE (2008)
7. Murphy, G.C., Notkin, D., Sullivan, K.: Software reflexion models: bridging the gap between source and high-level models. SIGSOFT Softw. Eng. Notes 20 (1995)
8. Sangal, N., Jordan, E., Sinha, V., Jackson, D.: Using dependency models to manage complex software architecture. In: OOPSLA (2005)
9. Terra, R., Valente, M.T.: A dependency constraint language to manage object-oriented software architectures. Softw.: Practice and Experience 39(12) (2009)
10. Herold, S.: Checking architectural compliance in component-based systems. In: SAC (2010)
11. Knodel, J., Muthig, D., Haury, U., Meier, G.: Architecture compliance checking - experiences from successful technology transfer to industry. In: CSMR (2008)
12. Knodel, J., Muthig, D., Naab, M., Lindvall, M.: Static evaluation of software architectures. In: CSMR (2006)
13. Rosik, J., Le Gear, A., Buckley, J., Ali Babar, M.: An industrial case study of architecture conformance. In: ESEM (2008)
14. Wong, S., Cai, Y., Kim, M., Dalton, M.: Detecting software modularity violations. In: ICSE (2011)
15. Bauer, C., King, G.: Hibernate in Action. Manning Publications Co. (2004)
16. Vespucci, http://www.opal-project.de/vespucci_project
17. Eichberg, M., Kahl, M., Saha, D., Mezini, M., Ostermann, K.: Automatic incrementalization of prolog based static analyses. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 109–123. Springer, Heidelberg (2007)
18. Simon, H.A.: The architecture of complexity. In: Proceedings of the APS (1962)
19. Koschke, R., Simon, D.: Hierarchical reflexion models. In: WCRE (2003)
20. Abi-Antoun, M., Aldrich, J.: Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. In: OOPSLA (2009)
21. de Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: A survey. Journal of Systems and Software 85(1) (2012)
22. Hello2Morrow Sotograph, http://www.hello2morrow.com/products/sotograph (accessed October 2012)
23. Hou, D., Hoover, H.J.: Using scl to specify and check design intent in source code. IEEE Trans. Softw. Eng. 32(6) (2006)
24. Gasparis, E., Nicholson, J., Eden, A.H.: Lepus3: An object-oriented design description language. Diagrams (2008)
25. Mens, K., Kellens, A., Pluquet, F., Wuyts, R.: Co-evolving code and design with intensional views. Comput. Lang. Syst. Struct. 32(2-3) (2006)
26. Morgan, C., De Volder, K., Wohlstadter, E.: A static aspect language for checking design rules. In: AOSD (2007)
27. de Moor, O., Sereni, D., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Tibble, J.: .QL: Object-Oriented Queries Made Easy. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 78–133. Springer, Heidelberg (2008)
28. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)

29. Siff, M., Reps, T.: Identifying modules via concept analysis. 25(6), 749–768 (1999)
30. Jacobson, I., Ng, P.W.: Aspect-Oriented Software Development with Use Cases. Addison-Wesley (2004)
31. Garlan, D., Barnes, J., Schmerl, B., Celiku, O.: Evolution styles: Foundations and tool support for software architecture evolution. In: WICSA/ECSA (2009)
32. Breivold, H., Crnkovic, I., Eriksson, P.: Analyzing software evolvability. In: COMPSAC (2008)
33. Aoyama, M.: Metrics and analysis of software architecture evolution with discontinuity. In: IWPSE (2002)

Towards Reactive Programming for Object-Oriented Applications

Guido Salvaneschi and Mira Mezini

Software Technology Group
Technische Universität Darmstadt
lastname@informatik.tu-darmstadt.de

Abstract. Reactive applications are difficult to implement. Traditional solutions based on event systems and the Observer pattern have a number of inconveniences, but programmers bear them in return for the benefits of OO design. On the other hand, reactive approaches based on automatic updates of dependencies – like functional reactive programming and dataflow languages – provide undoubted advantages but do not fit well with mutable objects.

In this paper, we provide a research roadmap to overcome the limitations of the current approaches and to support reactive applications in the OO setting. To establish a solid background for our investigation, we propose a conceptual framework to model the design space of reactive applications and we study the flaws of the existing solutions. Then we highlight how reactive languages have the potential to address those issues and we formulate our research plan.

Keywords: Reactive Programming, Functional-reactive Programming, Object-oriented Programming, Incremental Computation.

1 Introduction

Most contemporary software systems are reactive: Graphical user interfaces need to respond to the commands of the user, embedded software needs to react to the signals of the hardware and control it, and a distributed system needs to react to the requests coming over the network. While a simple batch application just needs to describe the algorithm for computing outputs from inputs, a reactive system must also react to the changes of the inputs and update the outputs correspondingly. Moreover, there are more tight constraints on computation time, because reactive systems work in real-time and need to react quickly – within seconds or even milliseconds. When the reactive behavior involves non-trivial computations or large amounts of data, various optimization strategies, such as caching and incremental updating, need to be employed.

Object-oriented programming does not provide specific mechanisms for implementing reactive behavior, with two consequences. First, reactive behavior is usually encoded by using the Observer design pattern, whose drawbacks have been extensively highlighted in literature [13,54,52]. For example, the code responsible for update of outputs is usually tangled with the code changing the inputs. As a result, it becomes difficult to understand the computational relations between inputs and outputs and, thus,

S. Chiba et al. (Eds.): Transactions on AOSD XI, LNCS 8400, pp. 227–261, 2014.
© Springer-Verlag Berlin Heidelberg 2014

the intended behavior of the system. Second, the update functionality with the necessary strategies to achieve the desired performances must be implemented manually for each application. Such optimizations, however, introduce a lot of additional complexity, so that it becomes an act of balance between complexity and efficiency.

Various approaches aim to address different aspects of these issues. Event-driven programming (EDP) creates inversion of control to enable modularization of the update code [64,34,28]. Aspect-oriented programming (AOP) enables complete separation of the update concern, by specifying in the aspects the points where the update needs to be triggered [41,67,62,8]. The above approaches fit well with mutable objects, but retain some of the problems related to a programming style based on inversion of control, similar to the well-discussed problems of the Observer design pattern.

Declarative reactive approaches, most notably functional-reactive programming (FRP) [26] and reactive languages, like FrTime [13], Flapjax [54], and Scala.React [52], completely automate the update process. The developer specifies only how a changing value is computed from other values, and the framework ensures that the computed value is automatically updated whenever the inputs are changed. It is, however, not clear whether they can obsolete manual implementation of update code. FRP and reactive languages deal with update of primitive values, and may be too inefficient since they do not provide incremental update of complex structures. Incremental update is provided by other approaches such as LiveLinq [50], but they are limited to specific data structures. Also, declarative approaches based on the functional paradigm do not offer the advantages of typical object-oriented designs, including modularization and component reuse.

In summary, the current state of the affairs is rather disappointing: Developers implement reactive applications in the *comfortable* world of objects, at the cost of relying on programming models whose limitations have been known for a longtime. On the other hand, alternatives based on reactive programming offer an appealing solution, but do not succeed because they do not provide the necessary flexibility and do not integrate with the OO design.

In this paper, we propose a research roadmap to fill the gap between OO design and reactive approaches. Our vision is that the concepts developed by FRP and dataflow programming can be integrated with object-orientation to provide dedicate support for reactive applications in mainstream languages. This goal is challenging because reactive abstractions have been explored mainly in the functional setting or in special domains, like reactive data structures. The analysis presented in this paper provides a solid background and the first steps in our research plans are already ongoing. Our initial effort is the development of RESCALA [69], a programming language that integrates events and behaviors *a la* FRP. Thanks to the conversions between events and signals, with RESCALA, reactive abstractions can be easily introduced in OO reactive applications. Our experiments show a significant improvement in the design of reactive software using the functionalities of RESCALA.

In summary, we provide the following contributions:

- We characterize the design space of reactive applications and discuss the strategies that can be applied to implement reactivity.

- To understand the practical impact of each update strategy, we analyze the implementation of reactive behavior in several real-world OO applications. Our analysis highlights the drawbacks of traditional abstractions.
- We analyze the existing language solutions for reactive systems. We underline their limitations, and the key achievements to take into account in further research.
- We propose a research roadmap which addresses the issues found in the current approaches and has the ultimate goal of combining objects and reactive abstractions in a flexible and efficient language.

The paper is structured as follows. In Section 2, we analyze the design space of reactive software. In Section 3, we present an empirical evaluation of real-world OO reactive applications. Section 4 outlines a possible alternative and discusses other research solutions. Section 5 presents our research roadmap. This paper is an extension of previous work from the same authors [70,68].

2 Design Space in OO Languages

In reactive systems, the outputs of the program need to be updated based on changes of inputs and time. The ways of achieving this goal are however very diverse. In this section, we overview the possible update strategies and discuss the rationale of choosing them.

To make the discussion more clear, in Figure 1, we show a model of a reactive software: The outputs provided to the clients (the objects a and e) must reflect the current state of the inputs of the application (objects b, c, and d) according to certain transformations f and g. Objects can be composed: For example, the object a contains the references a_1 and a_2 to other objects (e.g. by storing them in fields). Dashed arrows model dependencies: Output objects are computed from certain input objects. To clarify how the model applies, consider a weighted graph used to compute a *derived* graph. The derived graph is composed of references to the edges exceeding a `MIN` weight value. In the model of Figure 1 the basic graph can be represented by the b object that contains the references b_1, b_2, and b_3 to the edges. The derived graph can be represented by a and contains the references a_1 and a_2 to the edges. Finally, f is the transformation that produces the derived graph a from the basic graph b by filtering the edges according to the `MIN` value, modeled by the c object. For each update strategy, we show in a Java-like language how the dependent graph is obtained.

2.1 On-Demand Recomputation

The most straightforward approach is to recompute the output values each time they are needed. For example, when they are requested by the user to generate a report, or when she refreshes a view. Similarly, in real-time computer games and simulation applications, it is common to recompute outputs automatically at certain intervals of time or simply as often as possible. In OO design, the typical example of this approach are methods returning values computed from the state of the object each time these methods are called.

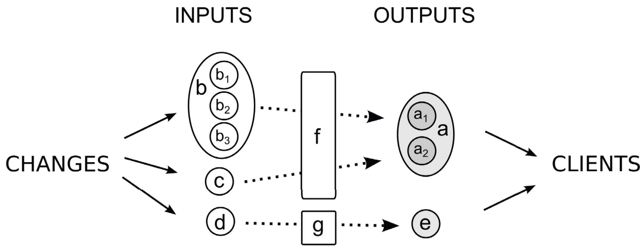


Fig. 1. A model of reactive behavior among objects

The distinguishing aspect of on-demand recomputation is that, after the evaluation, the output is discarded. For example, in Figure 1, every time a client requests a, a is recomputed and b is evaluated to calculate a. Figure 2(a) shows the on-demand recomputation strategy applied to the graph example: Every time the `getDerivedGraph` method is called, the dependent graph is computed from scratch by filtering all the edges (Line 6) and it is returned to the client.

The advantage of this approach is that it is simple to carry out, because the developer just needs to implement procedures computing the outputs from inputs. It also guarantees that the values are always computed from the current state of the program, and thus are always consistent with their inputs. The approach is also memory efficient, because only the inputs need to be stored, but not the outputs or any intermediate computation.

2.2 Caching

Recomputing outputs every time they are requested may be too inefficient, especially in the cases when the computations are expensive or need to deal with large amounts of data. Caching a computed result is a general optimization strategy that avoids repeating the computation. In the model of Figure 1, caching is obtained by saving a and e, letting them available for more than one client access. A typical design is to introduce a field for storing the computation results. The method that computes the dependent value is modified to return the value of the field if it is valid, and to compute and save the result otherwise. Figure 2(b) shows an implementation of the caching strategy: The `derivedGraph` is maintained in a field (Line 3) and returned only if valid, otherwise, the dependent graph is recomputed. When an edge is added to the base graph, the derived graph is invalidated (Line 17).

The cached values are valid only as long as the inputs of the computation do not change. When the inputs change, the cached value must be either recomputed, or invalidated and recomputed at the next request. The latter approach is more efficient when the computed value is used not so frequently, but it is also slightly more complicated. A major issue is to detect changes of the inputs and decide which cached values need to be invalidated. A straightforward approach is to invalidate all cached values after the change of every input. An efficient solution is to analyze the actual dependencies between inputs and outputs, and, after a change to an input, update only the outputs that depend on it – as we explain hereafter.

```
 1 class Graph {
 2  Edge [] edges;
 3
 4  getDerivedGraph(){
 5    Graph g = new Graph();
 6    for (Edge edge : edges){
 7      if (edge.weight > MIN)
 8        g.add(edge);
 9    }
10    return g;
11  }
12  ...
13 }
14
15
16
17
18
19
```

(a)

```
 1 class Graph {
 2  Edge [] edges;
 3  Graph derivedGraph;
 4  boolean valid;
 5
 6  getDerivedGraph(){
 7   if (!valid){
 8     derivedGraph=new Graph();
 9     for (Edge edge : edges){
10       if (edge.weight > MIN)
11        derivedGraph.add(edge);
12   } }
13   return derivedGraph;
14  }
15  addEdge(Edge e){
16    edges.add(e);
17    valid = false;
18  } ...
19 }
```

(b)

Fig. 2. On-demand recomputation (a) and caching with invalidation (b)

2.3 Tracking Dependencies

Instead of updating all outputs after a change to an input, the programmer can rather update only the outputs that actually depend on the changed input. For example, in Figure 1, a change in c requires an update of a, but e is still valid and should not be recomputed. A finer-grained tracking of dependencies can take into account that a only depends on the elements among b_1, b_2, and b_3 that exceed `MIN`. Figure 3(a) shows an implementation of dependency tracking with caching: The dependent graph is maintained in a field, and it is updated only when one of the edges with weight greater than `MIN` is added, i.e., the logic keeps track of the edges on which the derived graph depends. Figure 3(b) shows an implementation of dependency tracking with on-demand recomputation. In this case, the dependent graph is recomputed on every client request. The knowledge about dependencies is maintained by keeping edges in an ordered list (Line 2). In this way, the computation of the dependent graph can be performed by evaluating only a subset of the edges of the base graph. The evaluation is interrupted when it encounters an edge that is not part of the dependencies (Line 9).

Although tracking dependencies may seem straightforward, implementing this strategy in practice is usually not easy. The programmer needs a precise knowledge of computational relations between outputs and inputs. The dataflow of an application is usually not explicit in imperative code, and a careful code analysis is required to reconstruct it. Moreover, the actual dataflow of an application may depend on dynamic conditions (e.g., dynamic type of a variable in case of subtype polymorphism) and thus

```
1  class Graph {
2   Edge [] edges;
3   Graph derivedGraph;
4
5   getDerivedGraph(){
6     return derivedGraph;
7   }
8   addEdge(Edge e){
9     edges.add(e);
10    if (e.weight > MIN)
11      derivedGraph.add(edge);
12  } ...
13 }
```

(a)

```
1  class Graph {
2   List<Edge> orderedEdges;
3
4   getDerivedGraph(){
5     derivedGraph = new Graph();
6     for(Edge edge:orderedEdges){
7       if(edge.weight > MIN)
8         derivedGraph.add(edge);
9       else break;
10    }
11    return derivedGraph;
12  } ...
13 }
```

(b)

Fig. 3. Tracking dependencies with caching (a) and tracking with on-demand recomputation (b)

may be impossible to know statically. Developers must implement the update functionality that corresponds to the detected computational dependencies: After a change of each different input, the update of the corresponding outputs must be called. This may introduce a substantial amount of additional code. The update functionality may also cause modularity problems, because, when implemented in a straightforward way, it may introduce undesired dependencies from inputs to outputs. To avoid such dependencies, the programmer may employ various callback mechanisms (e.g., the Observer pattern), but this further increases the complexity of the implementation.

2.4 Update Incrementalization

Completely recomputing a cached value each time it is invalidated may be too expensive, especially if this value is a complex data structure, such as an array or a graph. A common optimization, in that case, is to update the cached value incrementally depending on the changes of the input. In the model of Figure 1, update incrementalization is an optimization of the functions f and g. This kind of optimization applies in presence of caching: to make updates incremental, the entity to update must be available to receive the changes. Figure 4(a) shows an example of update incrementalization: When a change occurs, the derived graph is not recomputed from scratch but it is modified gradually by adding only the edges that satisfy the condition (Line 13).

The similarity between Figure 3(a) and Figure 4(a), deserves more discussion. In contrast to Figure 3(a), where the dependencies are tracked to notify a change to the dependent graph only when needed, in Figure 4(a) the dependent graph is *always* notified of the change, regardless of the weight of the added edge (Figure 4(a), Line 7). Of course, the edge is added to the dependent graph only if the condition on the weight is satisfied. Figure 3(a) shows that caching with dependency tracking already

```
1  class Graph {
2   Edge [] edges;
3   DerivedGraph derivedGraph;
4
5   addEdge(Edge e){
6     edges.add(e);
7     derivedGraph.add(e);
8   } ...
9  }
10 class DerivedGraph
11     extends Graph {
12  addEdge(Edge e){
13     if (e.weight > MIN)
14       this.add(e);
15  } ...
16 }
17
18
19
```

(a)

```
1  class Graph {
2   Edge [] edges;
3   Graph derivedGraph;
4   Changes [] changes;
5
6   getDerivedGraph(){
7     applyChanges();
8     return derivedGraph;
9   }
10  applyChanges(){
11    /* Update derivedGraph
12    based on changes.
13    Clean added and removed */
14  }
15  addEdge(Edge e){
16    edges.add(e);
17    changes.add(new Add(e));
18  } ...
19 }
```

(b)

Fig. 4. Update incrementalization (a) and change accumulation (b)

implies some form of incrementality because the dependent graph has to be maintained (caching) and updated selectively (dependency tracking). However, these strategies are conceptually independent, as shown in Figure 4(b), where dependency tracking is demonstrated without caching. The similarity between Figure 3(a) and Figure 4(a) shows that the strategies analyzed in this section are often coupled and isolating them for the sake of the explanation leads to artificial examples: Dependency tracking and incrementalization typically appear together in applications – the former being necessary to support the latter.

Incremental update requires more fine-grained analysis of the changes to the inputs. It is not sufficient to detect that a certain input has changed, but it is also necessary to get precise information about the change. In addition, the programmer must design algorithms to update the value incrementally after different kinds of changes to the inputs. For example, in case the derived graph is the Minimum Spanning Tree of the original graph, specific domain knowledge in graph theory is required to implement the update algorithm incrementally.

2.5 Accumulating Changes

Accumulating changes is an optimization of the computation of outputs from inputs (i.e., an optimization of f and g in the model of Figure 1). Changes are stored and applied to a cached output, so caching is subsumed by this strategy.

Accumulating changes also implies the incrementalization of the update. Indeed, incrementalization is required to combine the existing object with the incoming changes. Accumulation allows one to arbitrarily choose when to apply the stored changes. One extreme is every time a change occurs (no accumulation), the other is every time the client requests the output. If the update of a value is postponed until the client request, this strategy avoids redundant updates of rarely requested values. Yet, combining a lot of accumulated changes is more expensive and the response time increases. In some cases, like in databases, the update is postponed until the end of some logical transaction in the inputs. Figure 4(b) shows an example of change accumulation. The changes to the base graph are accumulated in the `changes` array (Line 4). When the client requests the derived graph, the changes are applied and the derived graph is returned (Lines 6–8).

Updating a value after accumulating changes is usually more complicated than updating a value after each primitive change, because it requires more sophisticated data structures to describe the accumulated changes and more sophisticated algorithms to implement the update. As a result, this strategy increases memory consumption. However, accumulating changes also offers opportunities for optimization. For example, some changes can cancel each other. Nevertheless, a complex domain-specific logic is usually required to take advantage of such cases.

3 Case Studies

To analyze the design issues of OO reactive software, we inspected four reactive Java applications. Our goal is not to develop a systematic empirical study on OO reactive software. Instead, we want to provide a solid background for our research by surveying concrete examples of how reactive features impact OO software design. Due to space reasons, we show only a summary of our analysis. The interested reader can find more details in the technical report [65].

The case studies are of different sizes and cover different kinds of software (two desktop applications, a mobile application, and a library) as well as a variety of external sources of reactive behavior, like network messages, data sampling, values from sensors and user input. Figure 5 summarizes the main metrics of each application.

The **SWT Text Editor** (the `StyledText` widget) implements a text editor in the popular SWT library used by the Eclipse IDE [23]. The application reacts to the insertion of characters and to the formatting commands from the user.

The **FreeCol Game** [32] is an open-source turn-based strategy game. The AI of the game controls the opponent players, so the application reacts to the user and the AI. Updates concern the game model and the map in the GUI.

Apache Jmeter [45] supports the performance assessment of several server types (e.g., HTTP). The user specifies a test plan by adding graphical elements to a panel. First, the application must react to changes in the test plan. Additionally, the application is reactive to network events, as the results of the test are visualized in real-time.

The **AccelerometerPlay** Android application is one of the example applications provided by the Android platform [3]. It displays a set of particles rolling on the screen. The inclination of the device is detected by the accelerometer and the particles are updated accordingly.

Case study	LOC	Types	Cycl. Compl.	LOC/Method	Methods/Type	Fields/Type
SWT Text Editor	9,227	48	4.77	17.39	10.64	4.75
FreeCol Game	170,597	1,175	2.60	10.67	5.77	2.11
Apache JMeter	90,704	1,081	1.84	8.39	7.19	2.13
AccelerometerPlay	460	4	2.00	10.73	4.00	8.00

Fig. 5. Main metrics for the case studies

3.1 Design Choices in the Case Studies

Different design choices concerning reactive behavior are motivated in the case studies by the design, size, and kind of software. Our analysis clearly indicates that different applications are better "served" by different points in the design space depicted in Section 2.

The SWT text editor adopts caching and dependency tracking to achieve good performance. It is a typical example of an object with an internal state, which changes in the process of interacting with the user. The editor is implemented in the conventional event-driven style and it is highly optimized to ensure low reaction time. To this end, a lot of fields are used to cache intermediate values, and a complex logic takes care of updating values only when needed.

The FreeCol game lies at the opposite side of the design space and mostly adopts the on-demand recomputation strategy. Since the game behavior is inherently complex, the major issue is managing complexity to keep the development time and the stability of the game within reasonable limits. So, design decisions reducing or at least limiting complexity are favored. A substantial part of the code implements computations of values that are used in the user interface or for AI decision making. Almost all of these values are computed every time they are requested by the user or by the AI.

In the AccelerometerPlay application particle positions are recomputed on-demand every time the screen is refreshed, and incrementalization is applied to efficiently update the positions after conflict resolution. The logic is summarized in Figure 6: Position update comes first, then conflicts with the screen border and conflicts among particles are resolved. Finally, the particles are displayed. Since the application must react quickly, the update logic is based on imperative changes and an iterative algorithm is used for conflict resolution.

JMeter is optimized to make the GUI fast and reactive, changing it in response both to the user and the test events. Due to the different nature of these sources of change, the optimization strategies slightly differ. Caching is used for graphic widgets when the same graphic interface is associated to multiple elements in the test plan and can be reused when the user switches from one element to the other. Incrementalization is applied to optimize updates of graphs and statistics displaying the results of an ongoing test.

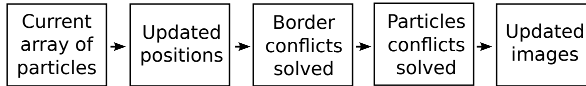


Fig. 6. The logic of the AccelerometerPlay application

3.2 Problem Statement

Our analysis revealed several design issues. We argue that these problems – commented hereafter – are not due to bad choices by the programmers. Instead, as we explain in Section 3.3, they are the consequence of the design limitations and the trade-offs imposed by OO language abstractions.

Code Complexity. Manually caching intermediate values requires an accurate logic that is responsible to perform the updates and maintain consistency. Performances increase, but the application becomes more complex. In the SWT text editor, the presence of a lot of fields in each class (70 mutable fields in the worst case) makes reasoning on the behavior of the application really hard, since computations depend on previous state. In addition, the update logic for enabling reactivity pervades a considerable part of the application. For example, the `StyledText` class includes 11 "addListener" methods, 11 "removeListener" methods, and 18 "handleEvent" methods. Moreover, the event-handling code includes anonymous classes created on the fly, which also expose callback methods. Finally, since values are separated from their update logic, local reasoning is impossible and understanding the application behavior requires inspecting a lot of code.

On the other hand, on-demand recomputation, like in the FreeCol game, clearly simplifies the logic of the application: As values are generated only when required, the behavior is not hidden by the code that maintains the dependencies.

Hidden Design Intent. The AccelerometerPlay application is an example of how reactive functionalities can hide the design intent of the developer. Although it is quite simple (less than 500 LOC), the reactive logic is spread all over the code and a conceptual model like the one in Figure 6 must be harvested from the system of callbacks and events. The origin of this complexity is that the design intent of each update strategy is not explicit in the implementation. For example, certain values are functionally dependent on other values but the design does not express this aspect. Only a careful analysis of the code reveals that a field is never changed directly, but updated after the changes of other fields. To reconstruct the intended computational dependencies, the developer must analyze all the update code scattered across the application. Our analysis revealed that computational dependencies are quite common in complex applications. For example, we determined that in the `StyledText` class of the SWT text editor, about half of the 70 mutable fields are not freely changeable, but store values functionally dependent on other fields. Despite that, all fields are declared in the same way and identifying dependent fields requires to reverse engineer the logic of the application, which is lost in the callbacks.

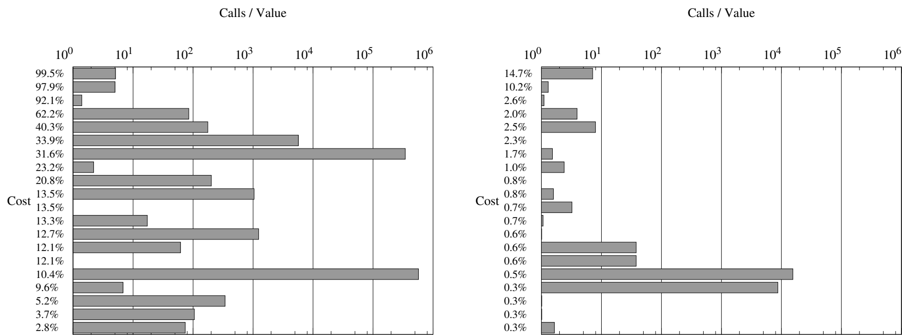


Fig. 7. Redundancy of the most expensive methods in the FreeCol game and the SWT text editor

Redundant Computations. The major advantage of on-demand computation is to keep the design simple. However, the overhead that is observed when this design choice is prevalent can be relevant.

We used a profiler and instrumentation via AspectJ to count the potentially redundant calls of the most time-consuming methods in the case studies. The computations after which the returned value does not change (for the same parameters) are potentially redundant. The impact of the design choices on redundancy can be seen in Figure 7. It compares the level of redundancy in the 20 most expensive methods of the FreeCol game and of the SWT text editor, which lie at the opposite positions in the design space. Redundancy is measured as the number of calls per different observed values, i.e., if a method is called 10 times in total and only 2 different input values and return values are observed, the redundancy is 5. The SWT text editor (right), thanks to its complex logic, shows values of redundancy which are substantially lower. We devoted further investigation to the potential optimizations in the FreeCol game, which largely adopts on-demand recomputation. The results for the methods with higher relative time are shown in Figure 8 (the percentages in the first column do not sum to 100% because methods can be nested). For example, we discovered that the most expensive method (99.57% of the time) has 80% of potentially redundant calls, i.e., the computed value changes only in $\sim$26% of the method calls (Figure 8).

Scattering and Tangling of Update Code. When values are intermediately cached, they must be updated in every point of the application where the inputs of the computation are changed. This leads to scattering of the update code.

To evaluate code scattering in the case studies, we considered the places where a field is directly written except the initialization. Figure 9 shows which percentage (y) of fields is updated in x places for each application. The analysis shows that most of the fields are updated only a few times, but the distribution has a *heavy tail*, i.e., there is a consistent number of fields that are updated in many places. Not surprisingly, the SWT text editor, which adopts the caching strategy and is highly complex, has the highest number of fields updated in several places. In the SWT text editor, setters are not used extensively, presumably because they are supposed to be used by the clients of the

Relative Time	Call / Value	Average Change	Calls	Time
99.57%	5.1	0.2650007264	14,798	76,705
97.98%	5.0	0.2672708364	14,732	75,821
92.13%	1.4	3.8125	29	36,216,475
62.20%	85.0	0.0009699462	8,507,562	83
40.35%	176.5	0.0038796778	5,437,321	84
33.97%	5,697.2	0	28,378,977	13
31.60%	344,846.8	0.1260744986	13,360,833	26
23.23%	2.2	0.0018647649	2,971,606	89
20.82%	201.8	0.0260969848	10,367,994	22
13.56%	1,041.0	0	1,579,745	97
13.56%	1.0	0	5,626	27,471
13.30%	17.3	0.6696185286	6,482	23,392
12.71%	1,241.2	0	1,206,869	120
12.13%	61.9	0	5,451	25,371
12.13%	1.0	0	5,451	25,365
10.49%	571,769.7	0	22,149,669	5

Fig. 8. Redundancy analysis for most expensive methods in the Freecol game

library but are avoided inside the library for efficiency reasons. Even so, since it is a best practice to implement setters and getters to encapsulate state, we repeated the analysis for setter methods in Figure 10. With the exception of the AccelerometerPlay application which is too small to suffer from scattering and tangling issues, we found that update scattering is extremely common. For example, in the FreeCol game, in JMeter and in the SWT text editor, respectively, only 38.4%, 46.0%, and 30.7% of the fields is updated in just one place. In the worst case, a field was updated in 96 places!

Error-Proneness and Code Repetitions. When the updates of the dependencies are managed manually, it is often hard for developers to understand when to trigger an update. For this reason, programmers code in a defensive way and introduce an update even when not necessary. For example, in the JMeter application we found cases in which selecting a GUI with the cursor triggers a sequence of four updates of the interface even before the user changes any value. Yet, update errors are not the only inconvenience that manual updates can cause. When update functionalities are complex, managing consistency by hand can easily lead to code repetitions because the same update pattern is cloned in many places. We evaluated the occurrence of code similarities in the case studies; the results are in Figure 11. The numbers in the table show that

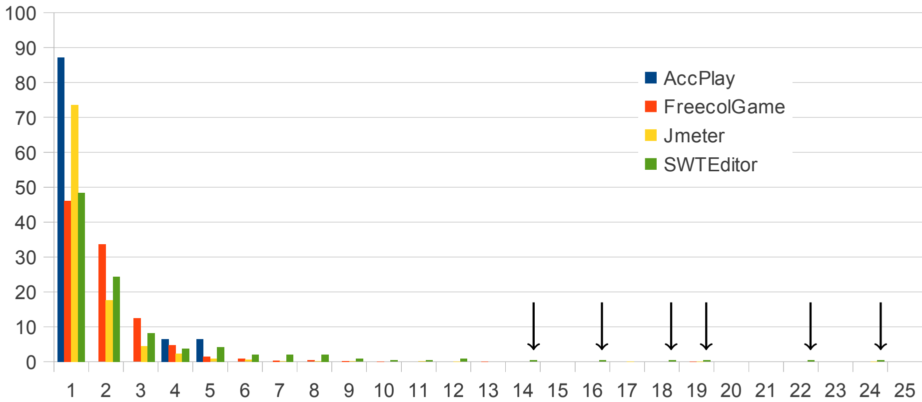


Fig. 9. Percentage of fields against number of different update places for that field

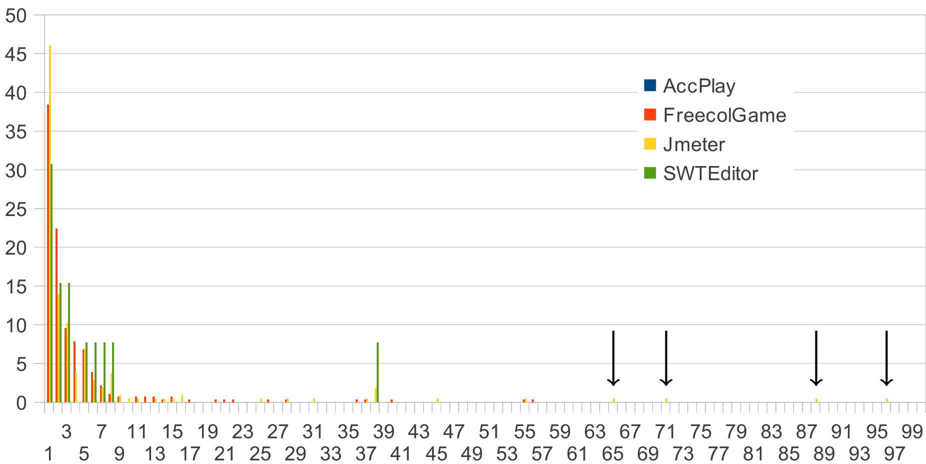


Fig. 10. Percentage of setter methods against number of different call places for that setter

for applications of significant size, even when the update functionalities are carefully designed, like in the case of the SWT text editor, it is hard to keep the code clean.

3.3 Lesson Learned from the Case Studies

In this section, we summarize the major results from the case studies. A crucial observation is that, in OO applications, reactive entities are separated from the code responsible to keep them updated. This has two bad consequences. First, the dependencies are not explicit, so the design rationale of the application is hard to grasp even for trivial cases. Second, updates are scattered across the application and tangle the rest of the code.

Unfortunately, modularization of update code is hard to achieve in the OO style, because dependencies must be imperatively updated every time an input value is changed

Application	Similarities	Similarities / LOC
SWT Text Editor	29	0.00314
FreeCol Game	281	0.00106
Apache Jmeter	381	0.00420
AccelerometerPlay	0	0

Fig. 11. Code repetitions

– which can occur in several places of the application. Furthermore, manually analyzing dependencies and writing corresponding update code is error-prone; certain dependencies may be overlooked and consequently the programmers can fail to update all functionally dependent values. Therefore, values are often updated defensively without precise knowledge of whether it is actually necessary.

Manually written update code also produces a maintenance problem, because there are no automatic checks ensuring the consistency of the update code with the actual dependencies of the computation. In addition, each time the computational dependencies are changed, the developer must correctly *update the update functionality* to reflect the current state of the dependencies! Errors of such a manual maintenance activity may remain undetected for a longtime; forgetting to update a certain value usually does not lead to a crash, and a redundant update may even not cause any visible effects at all, only inefficiency.

Importantly, in our analysis, we observe a clear trade-off between efficiency and complexity. To keep the design simple, programmers accept the cost of on-demand recomputation and potential redundancy. For example, intermediate caching via object fields highly complicates the application because the update logic must be implemented manually. In conclusion, keeping the design simple has a high cost in performance. In some cases, like the FreeCol game, there is a wide space for potential optimization. However, this is not easy to achieve, because the computations involve complicated algorithms and depend on various different inputs.

4 Analysis of Advanced Languages

In this section, we discuss some advanced language concepts to support reactive applications. For each approach, we analyze the problems it addresses and its limitations. We start the discussion with functional reactive approaches, as they provide interesting insights as how to overcome the problems discussed in the previous section and are the source of inspiration for our planned research, a roadmap of which is presented in the following section.

4.1 Reactive Languages

Reactive languages are based on ideas developed in functional-reactive programming, introduced by Elliott [26] to model time-changing values as dedicated language

abstractions. More contemporary incarnations of the concept are integrated in recent reactive languages such as FrTime [13], Flapjax [54], and Scala.React [52]. To make the argumentation more concrete, in Figure 12, we show an example of Flapjax, a dataflow language that overcomes several limitations of reactive design based on inversion of control. Our considerations can be substantially generalized to FrTime and Scala.React. The functionality presented in Figure 12 consists of displaying the elapsed time since the user clicked on a button in a Web page.

Flapjax supports behaviors, i.e., reactive abstractions that model time-changing values (named with a final "`B`" in the code snippet). For example, the value `nowB` is a behavior that represents the current time updated every second. Behaviors can be sampled to obtain "traditional" values via the `valueNow` function: `startTm` is the initial instant of the simulation. In addition, behaviors can be combined with events: In Line 3, the `snapshot` function captures the instant value of the `nowB` behavior every time the `click` event of the `reset` button occurs. As a result, `clickTmsB` always contains the time of the previous click (or `startTm` before any click event). `elapsedB` keeps the value of the time elapsed from the last click, and `insertValueB` updates the value in the graphic every time the `elapsedB` value changes (Line 6). The crucial aspect of the reactive semantics is that a declaration like the one in Line 5 expresses a *constraint* rather than a *statement*. The example shows how the language creates implicit dependencies among time-changing values. The general idea is that when the programmer defines a constraint `a=f(b)` and `b` is a time-changing value, the framework automatically detects the dependency of `a` on `b` and is responsible for performing the updates automatically. In Line 5, when either `nowB` or `clickTmsB` changes, the value of `elapsedB` is automatically updated. So, whenever the programmer accesses `elapsedB`, she sees the updated value. Reactive languages provide abstractions to compose time-changing values and combine them with event streams. Eventually, time-changing values are bound to the GUI which automatically reflects the changes. The reader interested in more details can refer to [54].

Automatic dependency tracking addresses several issues highlighted in the case studies. The application is simplified, because the programmer does not shoulder the burden of keeping dependent values consistent (Section 3.2). As a consequence, the errors that can derive from forgetting the updates are automatically avoided (Section 3.2). The update code, which captures the behavior of a program entity, is modularized with the entity, allowing local reasoning and avoiding scattering and tangling with the rest of the application (Section 3.2). In contrast to callbacks, which return void, reactive behaviors can be easily composed. Software is much more readable because the design intention of the programmer is explicit and direct modeling of relations among objects enforces a more declarative style (Section 3.2). Finally, reactive languages automatically derive dependencies and perform only necessary updates (Section 3.2).

In summary, reactive languages are an appealing solution to the issues identified for OO languages. In particular, since updates are performed by the runtime and do not add complexity to the application logic, they have the potential of solving the trade-off between efficiency and simplicity described in Section 3.3. Nevertheless, there are some crucial issues that prevent their broader adoption.

```
1  var nowB = timerB(1000);
2  var startTm = nowB.valueNow();
3  var clickTmsB = $E("reset", "click").snapshotE(nowB)
4                  .startsWith(startTm);
5  var elapsedB = nowB - clickTmsB;
6  insertValueB(elapsedB, "curTime", "innerHTML");
7
8  <body onload="loader()">
9    <input id="reset" type="button" value="Reset"/>
10   <div id="curTime"> </div>
11 </body>
```

Fig. 12. Automatic dependency tracking in Flapjax

Functional Flavor and Immutability. Reactive languages impose a functional style, while OO programming features an imperative style. When a lot of code already exists, a functional refactoring of the entire application is in general not acceptable. Some computations are cumbersome to express functionally, while retaining acceptable performance and algorithmic clarity. For example, the conflict resolution algorithm in the AccelerometerPlay application is expressed in an imperative style using for loops, sequences of imperative statements to detect the conflicts, and imperative updates of the particles positions. The AccelerometerPlay application is also an example of performance-critical software. Conflicts among potentially hundred of particles must be solved in a sufficiently short time that the movement appears fluid to the user. Expressing the resolution algorithm in functional style with somewhat acceptable performance, would involve accumulative recursion: Functions in this style are rather hard to understand and, yet, probably not as efficient as encodings based on loops and imperative updates.

A consequence of the functional flavor of reactive languages is that they are effective with primitive values, but do not fit well with mutable objects. Strategies for incremental computation are highly application-specific and a framework can hardly address the problem in a domain-agnostic way. Therefore, there is no way to automatically incrementalize object updates. So, reactive languages recompute the dependent object every time the base object changes. To clarify this point, consider the expression `list2 = list1.filter(x>10)`, an instance of the case `a=f(b)` – discussed previously – where `a` and `b` are `list2`, respectively `list1` and `f` is the `filter` function for the given predicate. The expression establishes a dependency between `list1` and `list2` via `filter`. In our case, each time object `list1` changes, the `filter` operator produces a new `list2` object. Instead, imperative approaches update the mutable dependent object in-place, which is more efficient and preserves the object identity. For example, the SWT text editor employs mutable data structures to efficiently store the inserted text.

These observations are symptoms of a more general problem in reactive languages: The update strategy is hardcoded in the reactive framework, so only a point in the design space described in Section 2 is available to the programmer. As a consequence, efficiency might be an issue even for trivial cases.

Design of Complex Systems. So far, we defended the mutability of OO. Another, even more important reason, why we do not want to abandon the OO style is that it has established itself as the paradigm of choice for complex applications, for reasons related to design clarity and evolvability. A first remark is about modeling: Objects are effective in modeling complex systems because they reproduce the interaction of real-world entities. For example, the hundreds of simulation elements used by the FreeCol game are conveniently represented by objects.

Objects enable the development of large applications by modularizing rather large pieces of functionalities while abstracting over implementation details. For example, all the SWT widgets are ready-to-use components, but at the same time open for future modifications via subtype polymorphism: E.g., the SWT text editor uses a default implementation for the text container, but clients can provide a custom container by implementing a proper interface.

OO also supports reuse by class inheritance. This aspect is crucial in libraries which are developed incrementally, e.g., the `StyledText` class is part of an inheritance hierarchy of depth 7. Another key requirement of complex systems is runtime variability. Objects address this issue via dynamic polymorphism. For example, the simulation elements in JMeter are treated uniformly and late bound depending on the user decisions. In the SWT library, a text editor can be used wherever a generic widget is expected.

4.2 Functional-Reactive Programming

As already discussed, functional-reactive programming has been introduced in the seminal paper [26]. Further approaches refined and extended the concept, mostly focusing on the formal semantics of continuous time [59]. This leads to an elegant formalization with a denotational semantics in which behaviors are modeled as functions from time to values [26].

The first FRP implementations were pull-based, i.e., reactive values are pulled periodically by the reactive runtime to update the entities depending on them, e.g. a graphic animation. This approach corresponds to our on-demand recomputation (Sec. 2.1) except that the runtime performs the requests, while in Sec. 2.1 we expect that the client request a result when needed. Note that this model is different from reactive languages like FrTime, Scala.React and Flapjax which are push-based: The dependency graph is updated only when a reactive value changes. Push-pull FRP [27] is an attempt of a mixed model that leverages the efficiency advantages of push-based implementations. However, exploring the trade-offs between those models and evaluating their impact in practice is still an open problem (Section 5.4).

FRP has been successfully applied to several practical scenarios, including coordinating robots [43], network switches programming [31], and wireless sensor networks [58]. Current work focuses on optimization and safety guarantees enforced by the type system. For example, Krishnaswami *et al.* [49] use linear types to control allocation of new nodes in the dependency graph and avoid memory leaks.

4.3 Observer and Event-Driven Programming

The Observer pattern enables decoupling the code that changes a value from the code that updates the values depending on it. The Observer pattern has a number of drawbacks that have been extensively analyzed by researchers. The main points of criticism are summarized hereafter; the interested reader can refer to [54,52] for a detailed discussion. First, with the Observer pattern, applications are harder to understand because the natural order of dependencies is inverted. A lot of boilerplate code is introduced to correctly implement the pattern. Another problem is that, since callbacks do not return a type but perform imperative updates on the objects state, reactions do not compose. Finally, the notification to the observers is triggered in an imperative fashion and can be easily scattered and tangled with the rest of the application. Interestingly, many of these points of criticism clearly emerged in our case studies.

Event-based languages like C# [14], Ptolemy [64], EScala [34], JEScala [74], and EventJava [28] directly support events and event composition as language constructs. These languages reduce the boilerplate code introduced by the Observer pattern and provide advanced features like quantification over events, event combination, and implicit events. Event-based languages integrate well with the imperative paradigm: The callback defines all the operations required for the update of the object state. As a result, this approach preserves object identity – opposed to functional solutions which necessarily compute a different object – and supports efficient fine-grained changes of the updated object.

The main drawback of event-based programming is that a significant amount of reactive dependencies are still modeled by encoding them in a programmatic way, rather than being supported by the language. The update functionality must be designed and implemented explicitly in the callback method. Caching must be managed manually, deciding a proper policy and coding it in the callback as well. Similar considerations apply for any optimization strategy. For example, accumulation of changes must be entirely implemented by the programmer. Another issue is that, since all the update functionality must be coded manually, consistency is not automatically guaranteed. Instead, the developer must take care of correctly notifying all the entities which are functionally dependent. This leads to error-prone code with the risk of notifying *too rarely*, breaking functional dependency, or *too often*, defensively inserting unnecessary updates. Finally, non-functional design choices are hard-coded in the callback implementation: The developer has no option to choose among different non-functional trade-offs discussed in Section 2, such as caching or incremental updates.

4.4 Aspect-Oriented Programming

In the context of reactive applications, AOP can be used to intercept objects modifications and keep dependent entities updated. Since AOP supports proper modularization of crosscutting concerns, the update functionalities are separated from the code of the object. For example, the Observer pattern can be implemented in a modular way by using AOP techniques [41]. Other researchers proposed AOP languages to modularize complex relations among objects [67,62]. AOP integrates well with the imperative style and the mutability of object's state. A point of criticism is that AspectJ-like

pointcut-advice models, which dominate AOP design, can potentially break OO modularity. Nevertheless, it has been shown that pointcuts can be integrated with an event system preserving OO-style modular reasoning [34].

Finally, many limitations of event-based programming hold for AOP, too. Most noticeably, updates must be performed explicitly in the aspect code and dependencies are not automatically tracked as in reactive languages. In addition, composition of reactive behaviors is not easy to obtain, since aspects interactions are complex to master compared to expression composition in functional programming.

4.5 Reactive Collections

Reactive collections define functional dependencies among data structures (often expressed via SQL-like queries). The crucial point is that the framework keeps the dependent structures automatically updated when the basic ones change. Efficient incremental updates have been investigated by the database research community for a long time in the context of the view maintenance problem [12]. More recently, researchers introduced these solutions into programming languages, intercepting the updates via AspectJ [76] or using code generation techniques [66] to trigger the updates on the dependencies. Off-the-shelf libraries include LiveLinq [50] and Glazed Lists [37].

Reactive data structures share some design advantages with reactive languages. Computational dependencies are expressed explicitly, so the application is easier to read: *Queries* describe the functional relations between program entities in a declarative manner. Update functionalities are automatically derived – with little or no additional complexity. The framework is in charge of keeping the functional dependencies specified by the query constantly up to dated. Finally, reactive collections implement efficient update of various reusable operators by using incremental changes and caching to avoid redundant computation.

The main limitation of reactive data structures is that the approach is restricted to a specific domain – changing collections. These frameworks provide out-of-the-box reactive data structures, but they do not support automatic update of other types of objects. Another issue is that reactive collections do not automatically detect all computational dependencies, but only exploit a set of predefined ones. In general, it is not possible to specify a generic expression and leave the framework the responsibility of tracking all the dependencies. For example, the predicate in a `filter` operator (Section 4.1) is not part of the dependencies mechanism, so any change to the predicate remains undetected. Apart from providing custom indexes to speedup certain queries, reactive collections rely on hard-coded non-functional design choices depending on the internal implementation. The programmer cannot fine-tune the update strategy or customize the caching behavior. Finally, most of these frameworks come in a very relational flavor. This further limits their integration with OO languages in which they are usually embedded.

4.6 Constraint-Based Languages

Constraint-based languages, like Kaleidoscope [33] integrate constraints among variables into OO languages. The focus of constraint-based languages is on expressing

relations among time-changing values in the program. However, this can lead to a rather different programming model compared to traditional OO or functional programming. For example, Kaleidoscope supports constraints that the framework attempts to enforce according to a priority ranking. Clearly, prioritizing constraints introduces a semantics that is likely unexpected for programmers, because some constraints can be never satisfied.

In contrast to reactive languages, that focus on composition of time-changing values, constraint-based languages focus on relations among values. As such, constraint-based languages typically neither support combinators for time-changing values nor integrate time-changing values and events. One-way constraints have proved effective in the scope of graphical interfaces: The Lisp-based Garnet [56] and Amulet [57] graphical toolkits support automatic constraint resolution to relieve the programmer from manual updates of the view. The design of one-way constraints is similar to the basic functionalities of reactive languages as supports the same limitation w.r.t. state and the OO model. In contrast to most reactive langauges, Garnet and Amulet also support cyclic constraints. A constraint in a cycle is evaluated at most once. If it is asked to evaluate a second time in the same constraint resolution phase, it simply reevaluates to the previous value.

SuperGlue [53] is an OO language that supports behaviors at the component level. Similarly to constraint-based languages, behaviors in SuperGlue are declarative constraints used to connect (*glue*) software components. A prioritization mechanism similar to the one of Kaleidoscope is also available. A peculiar aspect of SuperGlue is that constraints can include quantification over sets of time-changing values. Quantification is expressed using types to refer to a set of variables. It worth noticing that quantification has been independently explored by different approaches to reactive programming as a mean to decouple software components. Examples include AOP, event-based languages like Ptolemy and constraint-based languages like SuperGlue.

4.7 Synchronous Dataflow Languages

In synchronous dataflow languages like Lucid [63], the program defines a network in which a synchronous signal propagates and triggers the computations in the nodes. Lustre [38,11] and SIGNAL [35] are examples of dataflow languages fostering a declarative style in which program and specification coincide. In those languages absence of cycles is enforced by construction to avoid inconsistencies. Esterel [4] is a similar approach but fosters has a more imperative style. In Esterel, cycles are checked by the compiler and rejected statically. Synchronous dataflow languages address the problem of specifying, programming and verifying reactive real-time systems. They enforce a programming model where the reaction time of the systems is virtually zero (synchrony hypothesis). In practice, the reactive system must be proved fast enough compared to the environment. For this reason, synchronous dataflow languages are usually compiled into easily verifiable models, e.g., finite state machines. Guarantees of real-time and memory-bound execution are provided at the cost of limiting the language expressively. A common example of such limitations is relinquishing higher-order behaviors.

A related family of languages are graphical languages like LabVIEW [46] for data acquisition (e.g., from an external board), signal elaboration and industrial control,

and Simulink [17] used for system simulation especially in control theory. The application domain of those languages makes the dataflow computational model particularly suitable and the graphical approach simplifies the use by non-programmers. Researchers have already explored synergies between graphical languages and synchronous dataflow languages. Tripakis *et al.* propose a method to translate Simulink to Lustre [73]. This approach allows model-based development of embedded systems software with Simulink, the *de facto* standard of many industrial applications domains. On the other hand, the advantages of Lustre are preserved, including formal semantics and static analysis, verification, and controller synthesis tools.

4.8 Self-adjusting Computation

Self-adjusting computation [2] studies the automatic derivation of incremental programs from batch ones. This solution adopts an algorithmic approach, focusing on reducing the computational complexity of the incremental application. Self-adjusting computation has been introduced in the functional setting [2]. More recent approaches focus on applying similar concepts to imperative languages [1,39] and low-level languages like those used in stack-based abstract machines [40].

In contrast to reactive languages, which focus on explicitly modeling time-changing values using proper language abstractions, self-adjusting computation focuses on deriving an incremental version of a batch algorithm. More generally, self-adjusting computation aims at speeding up algorithms by taking advantage of incrementality. However, self-adjusting computation does not provide new abstractions to express time-changing values. As a result, it is not clear how incremental programs can be integrated into existing applications (e.g., by interfacing time-changing values with events) nor how incremental programs can support modular composition. Instead, reactive languages intentionally enforce a programming model that supports time-changing values and their composition to achieve better design. Despite the different focus, we envisage important synergies between the two approaches. In particular, the solutions explored by self-adjusting computation can be applied to generic computations, solving in an automated way the problem described in Section 2.4. For this reason, this technique can be used to incrementalize the updates performed in reactive languages.

Recent work on self-adjusting computation merges ideas from incremental computing and programming models for Big Data, like MapReduce [18]. Incoop [5] is an incremental MapReduce framework. When Incoop detects changes to the input dataset it obtains the result of the computation through a fine-grained update of previous results. A fundamental feature of Incoop is that it is *transparent* to the user, i.e., the application is not different from a traditional MapReduce interface and all the incrementalization is done under the hood.

4.9 Complex Event Processing

Complex event processing (CEP) is about performing queries over time-changing streams of data. Event streams are combined and correlated to define *complex* events triggered when a correlation condition is satisfied. Typical application scenarios include intrusion detection [16], stock trading [19], and power management [78].

CEP has similarities with databases. Event streams in CEP can be thought as time-changing database tables which are updated every time a new event occurs (i.e., a new entry in the table is added). In contrast to database software, where data change relatively rarely and the user triggers the evaluation of queries over the existing data, in CEP, data change dynamically and the queries must be reactively evaluated upon data arrival. Queries are expressed in terms of time windows to correlate only the events that are included in a given time frame [16]. Because of the similarity between recognizing an event pattern and deciding whether a word belongs to a formal language, the semantics of event correlation is often expressed in terms of finite state automata. This is the case, e.g., of SASE [77], Cayuga [19] and TESLA [15].

Similarly to reactive languages, CEP operates on event streams and CEP query languages are usually very declarative. CEP systems support a semantics that includes time, and is in general more expressive than the semantics of event-based languages, like EScala, supporting only event combination. In contrast to event-based languages, CEP systems are typically implemented as separate applications accessed through a proper interface. For this reason, CEP query languages are usually specified as strings – a state of the affairs that is similar to databases and SQL queries. As such, they do not take advantage of in-language integration, including safety guarantees from the compiler and integration with other language abstractions, as reactive languages do.

5 A Research Roadmap

In this section, we present a research roadmap for embedding direct support of reactive applications in object-oriented programming languages. The milestones in the roadmap are ordered from the basic ones – which are ongoing, like the integration of reactive abstractions into an event model, to more elaborate ones – which address complex systems that require automatic adaptation. Beside language design, we plan to work on the improvement of the performance of reactive languages. Previous work mostly focused on language abstractions, with less attention to optimization or performance assessment. It must be noted that the high overhead of current reactive languages is also among the factors that limit the spreading of this technology.

5.1 Integration with Event-Based Programming

Languages with support for event-based programming do make an important step forward in more directly supporting reactive behavior in an imperative object model. As such, they are an ideal starting point for our research. Yet, as argued, they lack the capability of declaratively expressing reactive computations dependent on changing values. In summary, both events and reactive expressions are needed. Events support fine-grained updates of mutable objects. Reactive abstractions capture reactive computations in a compact and declarative way.

Hence, a first step in our plan is to seamlessly integrate reactive abstractions into object-oriented event systems. This goal requires the design of the interface between the reactive abstractions and the abstractions for imperative events, such that they can

be treated uniformly in computations and become composable. Such interface is fundamental to support a mixed programming approach and gradual migration of existing software to a more functional and declarative style.

A further step is to integrate reactive abstractions in other aspects of the language. For example, collections must react to changes of the contained elements. It has been shown that changes can be suitably provided to the clients via an event-based interface[1]. Similarly, reactive abstractions can be conveniently used in data structures to model properties which are functionally dependent on other values (e.g., the size or the head of a list, both functionally dependent on the content of the list). The result is a library of *reactivity-enabled* data structures, which expose certain values as reactive abstractions.

This work is currently ongoing in the incarnation of RESCALA (R*eactive*EScala), a language which integrates the advanced event system of EScala [34] and time-changing values in the style of Scala.React [52]. Our experiments show that with RESCALA the design of reactive applications is improved according to several metrics, including number of composable abstractions and removed callbacks [69].

5.2 Integration with Object-Oriented Design

Reactive languages provide abstractions to represent time-changing values. For simplicity, we assume the Flapjax terminology established in Section 4.1 and we refer to these abstractions as behaviors. Behavior values are bound to expressions that capture the dependencies over other values. For example, in Figure 12, Line 5, the `elapsedB` value is bound to the behavior expression `nowB-clickTmsB`. It is unclear, however, how behaviors should integrate with OO design.

We believe that behaviors should be part of the interface of an object and clients should attach to public behaviors to build complex reactive expressions. Private behaviors should instead model functionally dependent values that are consumed only inside the object. Clearly, object encapsulation should be supported to hide implementation details from clients. These results can be trivially achieved by applying visibility modifiers to behaviors, but the next steps in the integration with OO design require more investigation.

An open question is whether behavior expressions can be reassigned. A negative answer leads to a design more similar to method bodies in most OO languages: They are statically defined at development time and at runtime can only be executed. On the contrary, modifiable behaviors imply a design similar to fields that can be accessed by getter and setter methods. In that case, behavior expressions would be changeable. Since behavior expressions capture the dependencies over other application entities, allowing their reassignment introduces a potentially excessive degree of dynamicity, especially if behavior expressions can be reassigned from outside the object. Nevertheless, in a large application, it can happen that the dependencies of a long-lived component are not known when the component is instantiated and, depending on the evolution of the system, must be assigned during its lifetime.

[1] For example, the .NET framework provides the `System.Collections.ObjectModel.ObservableCollection<T>` class which exposes to the clients the `PropertyChanged` and the `CollectionChanged` events.

While in the existing literature behaviors are usually assigned and not modified later, this mostly seems due to accidental circumstances rather than justified by design considerations. First, the use cases provided in literature are mostly small examples in which reassignment is not really needed. Second, the functional flavor of the existing solutions presumably favors single assignment. Third, reassignment complicates the reactive model both from an implementation and a semantic standpoint, so non reassignment has been favored also for the sake of simplicity. As a result, we still lack a broad discussion of these issues that concretely justifies the preference for a model or the other.

Another open issue concerns inheritance. Should it be possible to override a reactive value with a new dependency expression or refer to the overridden one via *super*? Intuitively, this seems desirable, but the consequences on the propagation model need careful investigation as well as the expected benefits. A final consideration is about polymorphism. We envisage a scenario in which reactive entities are late bound – like objects – and the dynamic type of the reactive value captures the dependencies over the other entities of the application.

In summary, while reactive abstractions have been applied in the context of OO languages before, previous approaches focused on reactive fragments that only superficially challenge OO design. Consequently, we still lack a systematic investigation of the interaction between OO features and reactive abstractions. A starting point for our work is [44] which focuses on the specific scenario of an application in a functional reactive style interfacing with an OO graphic library.

5.3 Efficient Reactive Expressions

Reactive languages enable to define arbitrary constraints on dependencies between objects and leave the framework shouldering the recomputation of the dependent objects. However, as shown in Section 4.1, current approaches enforce immutability and recompute dependent objects every time, which negatively impacts efficiency. On the other hand, reactive collections (Section 4.5) overcome this problem by applying advanced strategies such as update incrementalization for a predefined set of operators.

Unfortunately, optimizations are provided out-of-the-box for built-in operators and are not at the fingertips of end developers. As discussed in Section 2.4, in certain circumstances, only domain knowledge enables the developer to provide a mechanism that supports incremental updates. Hence, a predefined set of operators is not sufficient.

Motivated by the above observations, a fundamental step in our research roadmap is to design a framework that combines the open approach of reactive languages, which support arbitrary reactive computations, with the efficiency of built-in reactive data structures. This solution overcomes the frustrating state of the art, where efficient reactive data structures and reactive languages are separate worlds. We will follow two lines of research.

As a first step toward this goal, we aim at bringing efficient built-in operators to reactive abstractions. We will provide a variety of efficient operators that seamlessly operate on reactive collections and reactive abstractions. Highly efficient libraries will be designed along the lines of [76,66], but integrated with the abstractions of existing reactive languages. For example, by allowing behavior-like expressions in the predicate

of a filter operator or by modeling the result of the reactive operators as behaviors. Predefined operators must cover the most common applicative scenarios such as collections and relational operations.

The second step of research will aim at reconciling the openness of reactive languages with efficient reactive operators. This is only possible if the optimization of generic computations is available to application developers. Optimizing a particular step of a reactive computation process must be handy: Providing a faster version of a reactive computation must be as easy as – say – overriding an existing method.

Our idea is to separate the *creation* of dependent objects, which is performed from scratch, from the *maintenance* of objects. In the default case, both creation and maintenance are accomplished by complete recomputation, i.e. by applying the function that relates basic and derived objects, for example `filter` in the `list2 = list1.filter(x>10)` expression. However, the programmer can provide refinements for the maintenance case. Those refinements can be implemented by imperative algorithms or by taking advantage of domain-specific knowledge to efficiently update dependent objects. In this way, efficient event-based computations that apply the optimization principles well-known from the OO context can be conveniently hidden behind high level operators expressed in a functional style. Reactive objects are then connected by those operators to compose constraints. To further open the framework, the programmer should be able to refine existing operators with a more efficient version when better performance is needed. Finally, late binding can be leveraged to obtain the dynamic selection of the best operator (i.e., the best refinement for a set of types).

A second line of research concerns optimizations that must take into account a broader scope than single operators. For example, considerable performance improvement in relational expressions comes from reordering by anticipating selections and deferring joins. In addition, those optimizations must be performed, at least partially, at runtime, to allow cross-module analysis. Deeply embedded DSLs come with powerful interfaces to support custom optimizations for DSL expressions embedded into the host languages. For example, in LINQ, the developer is provided with the raw compiler output in the form of an – internally untyped – expression tree. Scala-virtualized [55] employs a similar approach, but fosters more typing guarantees. We will investigate the applicability of these techniques to optimizing reactive expressions. However, these mechanisms are quite low-level, optimizations are hard to perform and require highly specialized skills. It has been reported that building a LINQ provider for the RavenDB database took more time than building the database [24]. Also, they do not support dynamic optimizations. We will opt for hiding the complexity of those techniques behind higher level abstractions.

5.4 Propagation Model

To enforce the constraints defined by the user, reactive languages keep a runtime model of the dependencies in the application. Usually, this model is a directed graph in which a change in a node triggers an update over the transitive closure of the dependency relation. Reactive languages mostly enforce a push propagation model in which changes are proactively applied to dependents in the graph [13,54]. However, also lazy models with invalidation of the cached values and on-demand recomputation have been

proposed [52]. The propagation of the changes along the graph has a considerable performance impact.

Optimization techniques regarding the propagation model have been already proposed. For example, *lowering* is a technique that applies static analysis to collapse several reactive nodes in the graph into one [9]. As a result, the computation is moved from the reactive model to the usual (and more efficient) call-by-value system. The Yampa FRP framework employs a similar approach but merges the computations at runtime [59]. Based on the above observations, one research direction that we plan to follow concerns optimizations related to the propagation model.

First, alternative graph constructions can lead to observationally equivalent reactive models. Consequently, performance considerations can guide the choice. For example, as noted in [9], always collapsing the computations can lead to poor performance in certain cases. Consider the following code snippet from [9]. A time-consuming operation depends on an operation whose output rarely changes. The second operation, instead, depends on a frequently changing value:

```
(time-consuming-op
  (infrequently-changing-op frequent-emitter))
```

Consider the case in which this code results in three nodes: A source node `frequent-emitter`, an `infrequently-changing-op` node which depends on the first one, and a `time-consuming-op` node, which depends on `infrequently-changing-op`. Every time `frequent-emitter` emits a new value, `infrequently-changing-op` is executed. Since the outcome of `infrequently-changing-op` rarely changes, `time-consuming-op` is executed just a few times. Instead, if `infrequently-changing-op` and `time-consuming-op` are collapsed into the same node, `time-consuming-op` is executed at the same rate `frequent-emitter` changes its value. This effect is even more significant in languages like Scala.React, which apply collapsing of computations as the principal composition mechanism. In summary, collapsing should not be accepted or refused in its entirety. Code analysis or dynamic techniques must be applied to detect in which case each solution leads to the best results.

A second aspect of the propagation model that needs further investigation is the choice of a push-based implementation that is adopted by most reactive languages. According to the design space presented in Section 2, this solution favors caching over on-demand recomputation. The choice is motivated by a constraint: A push-based solution is necessary to guarantee that possible side effects in the reaction are really performed. However, change propagation does not always involve side effects. An optimization should explore the space between caching and on-demand recomputation and provide a convenient compromise. For example, reactive data structures dynamically switch to a caching strategy when the requests exceed a threshold [76].

This solution is quite simple and does not consider factors like the current machine load. For example, if the load is low, it may be convenient to recompute cached reactive values even if they are rarely requested. On the other hand, with heavy load, this strategy can further degrade the performances without providing significant benefits. More advanced approaches relying on concepts from control systems need to be

investigated. The latter have been, e.g., explored for parallel data structures to provide the best performance adapting to different machines, configurations, and workloads [22].

Finally, update propagation models are common to both reactive languages and event-based systems. The former use these models to propagate updates across dependencies, the latter to trigger dependent events [34]. Since these mechanisms are more and more used in programming languages, an obvious question is if those functionalities should be supported at the VM level. In previous work, starting from similar considerations, we investigated dedicated VM support for AOP [7]. Research work on runtime environments which natively support the concept of reactive memory was recently carried out at the OS level [20]. Implementing a similar approach in a managed environment which specifically supports propagation of changes across reactive entities is still a research challenge.

A second research direction is optimization by design. This approach is enabled by making the performance implications of the language abstractions explicit and leaving the choice in the hands of the programmer. Current reactive languages focus on expressivity rather than performance. In practice, the programmer has no clear control of how performance is affected by the design choices. Unlike reactive languages, dataflow languages, like Esterel and LUSTRE, intentionally limit the expressive power of the available abstractions to achieve memory and time-bound execution. Attempts to limit expressivity to improve performances have already been done in the reactive languages community. For example, real-time FRP [75] is a time-bound and space-bound subset of FRP. However, real-time FRP is a *closed language* [48], i.e., it is not embedded in a larger general-purpose language, which considerably limits the applicability of this approach.

In summary, programmers face a black-or-white choice: Relinquish performance for expressivity or abdicating abstraction for efficiency. Instead, reactive languages should incorporate reactive primitives that, at the cost of a reduced expressive power, have a high-performance profile. Static analysis or a dedicated static type system can ensure that those primitives are not combined with the rest of the reactive system in a way that cancels the performance improvement.

A starting point is to implement lexically scoped dependencies. In current reactive approaches, reactive dependencies are established dynamically. When, during the evaluation of an expression, a reactive value is found, the value is inserted in the dependency graph. This approach introduces considerable overhead. In fact, the evaluation is slowed down by the process of double-liking-dependent values with their dependencies. Similarly, when the value of a node changes, it must be unlinked from the nodes depending on it. This behavior is required to keep the graph updated, since dependencies can change dynamically. For example, the value of the expression `if(a) b else c` depends on either `b` or `c` on the bases of `a`. As a consequence, the structure of the graph is not fixed but must be continuously restructured to capture the current dependencies [52]. In contrast, lexically scoped dependencies are fixed. This results in regions in which the graph structure does not change, avoiding the computations required to keep the graph updated and improving performances.

5.5 Seamless Integration of Reactive Programming

Current reactive programming solutions show different degrees of automation in tracking computational dependencies. In languages like Scala.React, the user specifies time-changing values – signals, in Scala.React terminology – and binds them to a reactive expression. During the evaluation of the reactive expression, when a time-changing value is encountered, it is added to the set of values the signal depends on. Time-changing values are identified by a special type, e.g. `Var[Int]`, which is used by the framework to recognize them and establish the correct dependencies. Flapjax and FrTime are more flexible and support *transparent programming* [13]: when a function receives a reactive value as an actual parameter, the function is *transparently* lifted to an equivalent function that produces a time-changing value. Function lifting is adopted in most implementations of functional-reactive programming (e.g., [25]). However, all these approaches require that the programmer identifies the primitive reactive values manually, by assigning them a special type.

The solutions discussed so far have several disadvantages. First, they are error-prone. Developers can forget to mark some dependencies as reactive. In that case, dependent values are not correctly updated when a change occurs. Second, reactivity must be planned in advance by identifying all the sources of reaction. Also, this approach does not support a gradual migration to a reactive style: existing applications require considerable effort to manually identify all the sources of change and wrap them with reactive types. An approach that automatically identifies the values a reactive value depends on can solve the aforementioned issues. In such a solution, when an expression marked as reactive is evaluated, it implicitly adds to its dependencies all the values that are read during the evaluation. When a change in one of those values occurs, the expression is automatically reevaluated.

A similar approach has been investigated by Ostermann *et al.* [60] in the context of advanced AOP languages. In [60], the authors present Alpha, a statically typed AOP language that demonstrates the role of an expressive pointcut mechanism in enhancing software modularity. In Alpha, pointcuts are Prolog [72] logic queries. Due to the expressive power of the pointcut language, it is possible to define a pointcut that captures all the write operations to the fields that were evaluated during a previous computation. In this way, when one of these values changes, the computation can be triggered again to obtain the new result. Alpha is an experimental language based on an ad hoc interpreter. An open problem is to make such a functionality efficiently available in mainstream languages. A first approach requires to implement a static analyzer that detects all the fields read inside a control flow. Write accesses to those fields are then intercepted to detect any change.

Beside efficient implementation, seamless integration of reactive programming requires to investigate some methodological aspects. For example, it can be an invaluable resource to refactor existing software to a reactive style. However, if the computation is repeated every time that an input changes, side effects are played again – a behavior that is typically undesired. For this reason, the migration to reactive programming also requires to refactor the application to move side effects out of signal expressions.

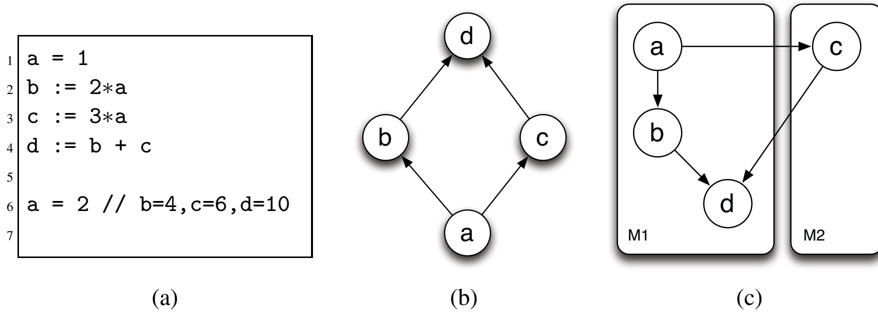


Fig. 13. A reactive program (a). The associated dependency graph (b). A possible distribution of the dependency graph over different hosts (c).

5.6 Distribution

Reactive applications include a wide class of distributed systems, like publish-subscribe systems [29], tuple spaces [36], and message-oriented middleware [42]. These systems typically use inversion of control to decouple the components of the application. As a result, they suffer from the shortcomings of the Observer pattern mentioned in Section 4.3. Our research roadmap includes exploring the use of reactive abstractions in the distributed setting. As a first step, we plan to make behaviors available remotely. Remote reactive objects are then similar to remote objects in Java RMI: clients can publish and lookup them through a public registry. In this scenario, the components in the distributed system expose time-changing behaviors to other components and build reactive computations by composing remote behaviors.

A sound design of such system is challenging. Reactive languages like Flapjax, FrTime or Scala.React organize the reactive values in a dependency graph in which each reactive object is connected to the objects it depends on. When a value in the graph changes, the change is transitively propagated to all the dependents to update them. The update order is controlled to enforce consistency properties in the propagation of the changes among behaviors. Consider the code in Figure 13(a) where `a`,`b`,`c` and `d` are behaviors and `:=` denotes a constraint among behaviors (not an assignment). When `a` changes, `b`, `c`, and `d` must be recomputed according to the dependency graph in Figure 13(b). If the evaluation order is `a-b-d-c`, `d` must be recomputed again to reflect the last change of `c`. Consequently, the expression that computes `d` is unnecessarily evaluated twice. More seriously, `d` is assigned an intermediate value – reflecting the change of `b` but not yet of `c` – that is *spurious* and only due to the propagation mechanism. Temporary inconsistencies of the propagation system during the propagation are referred to as *glitches*. Reactive languages typically enforce the absence of glitches (glitch-freedom) by updating the graph in topological order [51,13]. For example, the update order `b-c-d` guarantees glitch freedom.

Current reactive languages enforce glitch-freedom only on single hosts, but inconsistencies can still arise when inter-host communication is involved. Ensuring glitch-freedom in a distributed setting is not easy because the dependency graph is distributed

over different machines. In Figure 13(c) the graph spans over the machines M1 and M2. In this case, the propagation order is likely to be `a-b-d-c-d` because propagation on a single machine is probably faster than the one over the network, i.e., the message `a->b` is delivered before `a->c`. Applying the existing (local) algorithms to the distributed environment would require to pause the computation in all hosts and guide the change propagation across the distributed graph according to the topological order. However, this solution is clearly inefficient and requires a centralized manager that schedules the updates across the distributed system.

We envisage the following research directions. A first step consists of avoiding sequential updates in the graph. We plan to develop an algorithm for distributed reactive programming that maximizes the degree of parallelism among the hosts in the recomputation of the dependent nodes: independent nodes can be updated in parallel. This problem concerns parallelism *in the same propagation phase* along the graph. Another aspect concerns the parallelization of several update phases initiated by different hosts. A starting point is to consider the reactive values in the distributed reactive system as shared values in a concurrent system. An implementation can leverage distributed software transactional memories [71], which have been proved to efficiently scale up to large-scale clusters [6].

Web applications are a special case of distributed systems where only a client and a server interact. We expect that the implementation of a glitch-free propagation system can be optimized leveraging this simplification. For example, Javascript client-side code is single threaded, which greatly reduces the conflicts that can show up in the access to shared reactive objects.

Another research direction concerns the integration of the middleware that provides reactive programming with existing enterprise containers. In those systems, developers adopt conventions on the structure of certain objects and the container automatically enforces certain properties for them. For example, Enterprise Java Beans are automatically persisted by the container. We envisage a scenario in which, in a similar way, certain objects are automatically updated by the container thanks to the changes that propagate across the distributed dependency system.

Finally, an open research question is to find a balance in the trade-off between the properties enforced by the propagation system and performance. For example, in an environment where communications are not reliable, it can be reasonable to increase the efficiency of the propagation system at the cost of introducing glitches. The implementation of distributed reactive programming by Carreton *et. al.* [10] is a contribution in this direction focusing on mobile networks in the context of pervasive systems. In that framework, since the network can be extremely unreliable and hosts can experience temporary lack of connectivity, loose coupling is favored over safety and the propagation system in not glitch-free by design.

5.7 Evaluation

To make sure that our research leads to concrete results, we plan to continuously evaluate how it progresses. Evaluating language design is not easy, because design quality is hard to capture with synthetic metrics and design choices have long-term effects which are hard to predict. For example, the impact of programmers' experience or the

maintainability of large systems can be evaluated only when a considerable number of projects have been developed. As a consequence, we believe that a priori reasoning and careful analysis of the available options remain fundamental steps [47]. Nevertheless, where applicable, we plan to perform objective evaluations of our results.

Performance can be evaluated effectively by running benchmarks. For example, former studies compared the performance of OO programming (Java) with the mixed OO-functional style (Scala) in the context of parallel applications and multicore environments [61]. We believe that similar experiments can evaluate the performance impact of reactive abstractions compared to the traditional solutions for reactive applications.

Other aspects of the language design can be evaluated by using software metrics. Common metrics include coupling and cohesion, lines of code, number of operations and others [30]. We plan to adopt this approach to evaluate our design choices in the advanced state of our research, when studies can comprise several artifacts. Other researchers already used metrics to validate language design choices in reactive applications. Recently, the combined use of synthetic metrics and manual inspection (to investigate specific issues) was successfully applied to evaluate event quantification in software product lines [21].

Acknowledgments. This work is supported in part by the European Research Council, grant No. 321217 and by the German Federal Ministry of Education and Research (Bundesministerium für Bildung und Forschung, BMBF) under grant No. 16BY1206E “Sinnodium”.

References

1. Acar, U.A., Ahmed, A., Blume, M.: Imperative self-adjusting computation. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 309–322. ACM, New York (2008)
2. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM Trans. Program. Lang. Syst. 28(6), 990–1034 (2006)
3. Android developers web site, `http://developer.android.com/index.html`
4. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
5. Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U.A., Pasquin, R.: Incoop: MapReduce for incremental computations. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC 2011, pp. 7:1–7:14. ACM, New York (2011)
6. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2008, pp. 247–258. ACM, New York (2008)
7. Bockisch, C., Kanthak, S., Haupt, M., Arnold, M., Mezini, M.: Efficient control flow quantification. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, OOPSLA 2006, pp. 125–138. ACM, New York (2006)
8. Bodden, E., Shaikh, R., Hendren, L.: Relational aspects as tracematches. In: Proceedings of the 7th International Conference on Aspect-oriented Software Development, AOSD 2008, pp. 84–95. ACM, New York (2008)

9. Burchett, K., Cooper, G.H., Krishnamurthi, S.: Lowering: a static optimization technique for transparent functional reactivity. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2007, pp. 71–80. ACM, New York (2007)
10. Lombide Carreton, A., Mostinckx, S., Van Cutsem, T., De Meuter, W.: Loosely-coupled distributed reactive programming in mobile ad hoc networks. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 41–60. Springer, Heidelberg (2010)
11. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1987, pp. 178–188. ACM, New York (1987)
12. Ceri, S., Widom, J.: Deriving production rules for incremental view maintenance. In: Proceedings of the 17th International Conference on Very Large Data Bases, VLDB 1991, pp. 577–589. Morgan Kaufmann Publishers Inc., San Francisco (1991)
13. Cooper, G.H., Adsul, B.: Embedding dynamic dataflow in a call-by-value language. In: Sestoft, P. (ed.) ESOP 2006. LNCS, vol. 3924, pp. 294–308. Springer, Heidelberg (2006)
14. Microsoft Corporation. C# language specification. version 3.0 (2007), `http://msdn.microsoft.com/en-us/vcsharp/aa336809.aspx`
15. Cugola, G., Margara, A.: TESLA: a formally defined event specification language. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, DEBS 2010, pp. 50–61. ACM, New York (2010)
16. Cugola, G., Margara, A.: Processing flows of information: From data stream to complex event processing. ACM Comput. Surv. 44(3), 15:1–15:62 (2012)
17. Dabney, J.B., Harman, T.L.: Mastering SIMULINK 4, 1st edn. Prentice Hall PTR, Upper Saddle River (2001)
18. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
19. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.: Towards expressive publish/-Subscribe systems. In: Ioannidis, Y., Scholl, M.H., Schmidt, J.W., Matthes, F., Hatzopoulos, M., Böhm, K., Kemper, A., Grust, T., Böhm, C. (eds.) EDBT 2006. LNCS, vol. 3896, pp. 627–644. Springer, Heidelberg (2006)
20. Demetrescu, C., Finocchi, I., Ribichini, A.: Reactive imperative programming with dataflow constraints. In: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 407–426. ACM, New York (2011)
21. Dyer, R., Rajan, H., Cai, Y.: An exploratory study of the design impact of language features for aspect-oriented interfaces. In: 11th International Conference on Aspect-Oriented Software Development, AOSD 2012 (March 2012)
22. Eastep, J., Wingate, D., Agarwal, A.: Smart data structures: an online machine learning approach to multicore data structures. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, ICAC 2011, pp. 11–20. ACM, New York (2011)
23. Eclipse IDE Web site, `http://www.eclipse.org/`
24. Eini, O.: The pain of implementing LINQ providers. Queue 9(7), 10:10–10:22 (2011)
25. Elliott, C.: Functional implementations of continuous modeled animation. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 284–299. Springer, Heidelberg (1998)
26. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming, ICFP 1997, pp. 263–273. ACM, New York (1997)
27. Elliott, C.M.: Push-pull functional reactive programming. In: Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, pp. 25–36. ACM, New York (2009)

28. Eugster, P., Jayaram, K.R.: EventJava: An extension of Java for event correlation. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 570–594. Springer, Heidelberg (2009)
29. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
30. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F.C., Dantas, F.: Evolving software product lines with aspects: an empirical study on design stability. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 261–270. ACM, New York (2008)
31. Foster, N., Harrison, R., Freedman, M.J., Monsanto, C., Rexford, J., Story, A., Walker, D.: Frenetic: a network programming language. In: Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP 2011, pp. 279–291. ACM, New York (2011)
32. FreeCol game Web site, `http://www.freecol.org/`
33. Freeman-Benson, B.N.: Kaleidoscope: mixing objects, constraints, and imperative programming. In: OOPSLA/ECOOP 1990, pp. 77–88. ACM, New York (1990)
34. Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: EScala: modular event-driven object interactions in Scala. In: AOSD 2011, pp. 227–240. ACM, New York (2011)
35. Gautier, T., Le Guernic, P., Besnard, L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Kahn, G. (ed.) FPCA 1987. LNCS, vol. 274, pp. 257–277. Springer, Heidelberg (1987)
36. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
37. Glazed Lists Web site, `http://www.glazedlists.com/`
38. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proceedings of the IEEE 79(9), 1305–1320 (1991)
39. Hammer, M.A., Acar, U.A., Chen, Y.: CEAL: a C-based language for self-adjusting computation. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 25–37. ACM, New York (2009)
40. Hammer, M.A., Neis, G., Chen, Y., Acar, U.A.: Self-adjusting stack machines. In: Proceedings of the, ACM International Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA 2011, pp. 753–772. ACM, New York (2011)
41. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA 2002, pp. 161–173. ACM, New York (2002)
42. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java message service. Sun Microsystems Inc., Santa Clara (2002)
43. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Jeuring, J., Jones, S.L.P. (eds.) AFP 2002. LNCS, vol. 2638, pp. 159–187. Springer, Heidelberg (2003)
44. Ignatoff, D., Cooper, G.H., Krishnamurthi, S.: Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In: Hagiya, M. (ed.) FLOPS 2006. LNCS, vol. 3945, pp. 259–276. Springer, Heidelberg (2006)
45. JMeter developers Web site, `http://jakarta.apache.org/jmeter/index.html`
46. Johnson, G.W.: LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control, 2nd edn. McGraw-Hill School Education Group (1997)
47. Kiczales, G., Mezini, M.: Separation of concerns with procedures, annotations, advice and pointcuts. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 195–213. Springer, Heidelberg (2005)

48. Kieburtz, R.B.: Implementing closed domain-specific languages (Abstract of invited talk). In: Taha, W. (ed.) SAIG 2000. LNCS, vol. 1924, pp. 1–2. Springer, Heidelberg (2000)
49. Krishnaswami, N.R., Benton, N., Hoffmann, J.: Higher-order functional reactive programming in bounded space. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, pp. 45–58. ACM, New York (2012)
50. LiveLinq Web site,
`http://www.componentone.com/SuperProducts/LiveLinq/`
51. Maier, I., Odersky, M.: Deprecating the Observer Pattern with Scala.react. Technical report (2012)
52. Maier, I., Rompf, T., Odersky, M.: Deprecating the Observer Pattern. Technical report (2010)
53. McDirmid, S., Hsieh, W.C.: SuperGlue: Component programming with object-oriented signals. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 206–229. Springer, Heidelberg (2006)
54. Meyerovich, L.A., Guha, A., Baskin, J., Cooper, G.H., Greenberg, M., Bromfield, A., Krishnamurthi, S.: Flapjax: a programming language for Ajax applications. In: Proceeding of the 24th ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA 2009, pp. 1–20. ACM, New York (2009)
55. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. In: Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, pp. 117–120. ACM, New York (2012)
56. Myers, B.A., Giuse, D.A., Dannenberg, R.B., Kosbie, D.S., Pervin, E., Mickish, A., Zanden, B.V., Marchal, P.: Garnet: Comprehensive support for graphical, highly interactive user interfaces. Computer 23(11), 71–85 (1990)
57. Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A., Doane, P.: The Amulet environment: New models for effective user interface software development. IEEE Trans. Softw. Eng. 23(6), 347–365 (1997)
58. Newton, R., Morrisett, G., Welsh, M.: The Regiment Macroprogramming System. In: 6th International Symposium on Information Processing in Sensor Networks, IPSN 2007, pp. 489–498 (2007)
59. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, pp. 51–64. ACM, New York (2002)
60. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 214–240. Springer, Heidelberg (2005)
61. Pankratius, V., Schmidt, F., Garreton, G.: Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In: 2012 34th International Conference on Software Engineering (ICSE), pp. 123–133 (2012)
62. Pearce, D.J., Noble, J.: Relationship aspects. In: Proceedings of the 5th International Conference on Aspect-oriented Software Development, AOSD 2006, pp. 75–86. ACM, New York (2006)
63. Pouzet, M.: Lucid Synchrone, version 3. Tutorial and reference manual. Université Paris-Sud, LRI (April 2006)
64. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 155–179. Springer, Heidelberg (2008)
65. `http://www.stg.tu-darmstadt.de/media/st/publications/oo_reactive_report.pdf`
66. Rothamel, T., Liu, Y.A.: Generating incremental implementations of object-set queries. In: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE 2008, pp. 55–66. ACM, New York (2008)

67. Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., Komiya, S.: Association aspects. In: Proceedings of the 3rd International Conference on Aspect-oriented Software Development, AOSD 2004, pp. 16–25. ACM, New York (2004)
68. Salvaneschi, G., Drechsler, J., Mezini, M.: Towards distributed reactive programming. In: De Nicola, R., Julien, C. (eds.) COORDINATION 2013. LNCS, vol. 7890, pp. 226–235. Springer, Heidelberg (2013)
69. Salvaneschi, G., Hintz, G., Mezini, M.: REScala: Bridging between object-oriented and functional style in reactive applications. In: Proceedings of the 13th International Conference on Aspect-Oriented Software Development, AOSD 2014. ACM, New York (2014)
70. Salvaneschi, G., Mezini, M.: Reactive behavior in object-oriented applications: an analysis and a research roadmap. In: Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development, AOSD 2013, pp. 37–48. ACM, New York (2013)
71. Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing 10(2), 99–116 (1997)
72. Sterling, L., Shapiro, E.: The art of Prolog: advanced programming techniques, 2nd edn. MIT Press, Cambridge (1994)
73. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. Embed. Comput. Syst. 4(4), 779–818 (2005)
74. Van Ham, J.M., Salvaneschi, G., Mezini, M., Noyé, J.: JEScala: Modular coordination with declarative events and joins. In: Proceedings of the 13th International Conference on Aspect-Oriented Software Development, AOSD 2014. ACM, New York (accepted for publication, 2014)
75. Wan, Z., Taha, W., Hudak, P.: Real-time frp. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming, ICFP 2001, pp. 146–156. ACM, New York (2001)
76. Willis, D., Pearce, D.J., Noble, J.: Caching and incrementalisation in the Java query language. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications, OOPSLA 2008, pp. 1–18. ACM, New York (2008)
77. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD 2006, pp. 407–418 (2006)
78. Xiao, Y., Li, W., Siekkinen, M., Savolainen, P., Yla-Jaaski, A., Hui, P.: Power management for wireless data transmission using complex event processing. IEEE Trans. Comput. 61(12), 1765–1777 (2012)

Author Index

Akşit, Mehmet 27

Chiba, Shigeru 70

de Boer, Frank S. 1
de Gouw, Stijn 1

Eichberg, Michael 193

Figueroa, Ismael 145

Garcia, Alessandro 193

Johnsen, Einar Broch 1

Klein, Jacques 109
Kohn, Andreas 1

Macia, Isela 193
Malakuti, Somayeh 27
Mezini, Mira 193, 227
Mitschke, Ralf 193
Mouelhi, Tejeddine 109

Nain, Gregory 109
Nguyen, Phu H. 109

Salvaneschi, Guido 227

Tabareau, Nicolas 145
Tanter, Éric 145
Traon, Yves Le 109

Wong, Peter Y.H. 1

Zhuang, YungYu 70