

Alexander K. Petrenko  
Andrei Voronkov (Eds.)

LNCS 10742

# Perspectives of System Informatics

11th International Andrei P. Ershov Informatics Conference, PSI 2017  
Moscow, Russia, June 27–29, 2017  
Revised Selected Papers

 Springer

*Commenced Publication in 1973*

Founding and Former Series Editors:

Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

## Editorial Board

David Hutchison

*Lancaster University, Lancaster, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Friedemann Mattern

*ETH Zurich, Zürich, Switzerland*

John C. Mitchell

*Stanford University, Stanford, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*TU Dortmund University, Dortmund, 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, Saarbrücken, Germany*


More information about this series at <http://www.springer.com/series/7407>

Alexander K. Petrenko · Andrei Voronkov (Eds.)

# Perspectives of System Informatics

11th International Andrei P. Ershov Informatics Conference, PSI 2017  
Moscow, Russia, June 27–29, 2017  
Revised Selected Papers

*Editors*

Alexander K. Petrenko   
Ivannikov Institute for System  
Programming of RAS  
Moscow  
Russia

Andrei Voronkov  
The University of Manchester  
Manchester  
UK

ISSN 0302-9743 ISSN 1611-3349 (electronic)  
Lecture Notes in Computer Science  
ISBN 978-3-319-74312-7 ISBN 978-3-319-74313-4 (eBook)  
<https://doi.org/10.1007/978-3-319-74313-4>

Library of Congress Control Number: 2017964216

LNCS Sublibrary: SL1 – Theoretical Computer Science and General Issues

© Springer International Publishing AG 2018

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.

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.

The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Printed on acid-free paper

This Springer imprint is published by Springer Nature  
The registered company is Springer International Publishing AG  
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

## Preface

The Ershov Informatics Conference Perspectives of System Informatics (in the PSI Conference Series) is the premier international forum in Russia for research and applications in computer, software, and information sciences. The conference brings together academic and industrial researchers, developers, and users to discuss the most recent topics in the field. PSI provides an ideal venue for setting up research collaborations between the rapidly growing Russian informatics community and its international counterparts, as well as between established scientists and younger researchers.

The 11th edition of the conference was held during June 27–29, 2017, in Moscow (Russian Federation). Over 150 researchers and students participated in the event.

This volume contains the papers presented at PSI 2017. There were 57 submissions. Each submission was reviewed by at least two, and on average three, Program Committee members. The committee decided to accept 31 papers.

Dines Bjørner DTU (Denmark) spoke at the opening of the conference about the contribution of academician Victor Ivannikov to computer science, one of the organizers of this conference, who died at the end of 2016.

Famous scientists in the field of computer science presented invited talks:

- Sriram Rajamani (Microsoft Research, India) — “Trusted Cloud: How to Make the Cloud More Secure”
- Andrei Sabelfeld (Chalmers University of Technology in Gothenburg, Sweden and Gothenburg University, Sweden) — “Taint Tracking Without Tracking Taints”
- Michael Gerard Hinchey (Irish Software Engineering Research Centre, Lero) — “Building Resilient Space Exploration Systems”

We wish to thank all those involved in the support and organization of this conference as well as the Program Committee members and the anonymous reviewers. Without them and all their hard work, the realization of such an ambitious project would not have been possible.

December 2017

Andrei Voronkov  
Alexander K. Petrenko

# Organization

## Program Committee

David Aspinall	The University of Edinburgh, UK
Sergey Avdoshin	National Research University Higher School of Economics, Russia
Marcello M. Bersani	Politecnico di Milano, Italy
Eike Best	Universität Oldenburg, Germany
Nikolaj Björner	Microsoft Research, USA
Andrea Cali	University of London, Birkbeck College, UK
Mauro Caporuscio	Linnaeus University, Sweden
Gabriel Ciobanu	Romanian Academy, Institute of Computer Science, Iasi, Romania
Volker Diekert	University of Stuttgart, Germany
Salvatore Distefano	University of Messina, Italy
Nicola Dragoni	Technical University of Denmark, Denmark
Schahram Dustdar	Vienna University of Technology, Austria
Carlo A. Furia	Chalmers University of Technology, Sweden
Vladimir Galaktionov	KIAM RAS, Russia
Carlo Ghezzi	Politecnico di Milano, Italy
Sergei Gorlatch	University of Münster, Germany
Arie Gurfinkel	University of Waterloo, Canada
Cliff Jones	Newcastle University, UK
Joost-Pieter Katoen	RWTH Aachen University, Germany
Konstantin Korovin	The University of Manchester, UK
Maciej Koutny	Newcastle University, UK
Laura Kovacs	Vienna University of Technology, Austria
Gregory Kucherov	CNRS/LIGM, France
Anthony Widjaja Lin	University of Oxford, UK
Zhiming Liu	Southwest University, China
Rupak Majumdar	MPI-SWS, Germany
Manuel Mazzara	Innopolis University, Russia
Klaus Meer	TU Cottbus, Germany
Hernan Melgratti	Universidad de Buenos Aires, Argentina
Bertrand Meyer	ETH Zurich, Switzerland
Torben Mogensen	DIKU, Denmark
Peter Mosses	Swansea University, UK
Martin Nordio	ETH Zurich, Switzerland
Jose R. Parama	Universidade da Coruña, Spain
Wojciech Penczek	ICS PAS and Siedlce University, Poland
Peter Pepper	Technische Universität Berlin, Germany

Alexander K. Petrenko	ISP RAS, Russia
Qiang Qu	Shenzhen Institutes of Advanced Technology, China
Wolfgang Reisig	Humboldt-Universität zu Berlin, Germany
Andrey Rybalchenko	Microsoft Research, UK
Davide Sangiorgi	University of Bologna, Italy
Klaus-Dieter Schewe	Software Competence Center Hagenberg, Germany
Vitaly Semenov	ISP RAS, Russia
Natalia Sidorova	Technische Universiteit Eindhoven, The Netherlands
Mark Trakhtenbrot	Holon Institute of Technology, Israel
Irina Virbitskaite	A. P. Ershov Institute of Informatics Systems, of the SB RAS, Russia
Andrei Voronkov	The University of Manchester, Chalmers University of Technology, and EasyChair, UK/Sweedn
Domagoj Vrgoc	Pontificia Universidad Católica de Chile, Chile
Sergey Zykov	National Research University Higher School of Economics, Russia

## Additional Reviewers

Aman, Bogdan	Humernbrum, Tim	Robillard, Simon
Brown, Joseph Alexander	Höger, Christoph	Safina, Larisa
Chen, Wei	Junges, Sebastian	Schlachter, Uli
D'Angelo, Mirko	Kotelnikov, Evgenii	Silva, Alexandra
Dan, Li	Lange, Tim	Spina, Cinzia Incoronata
Erofeev, Evgeny	Li, Xiaoshan	Szreter, Maciej
Giallorenzo, Saverio	Ma, Hui	Tyszberowicz, Shmuel
Gonzalez, Senen	Mikulski, Lukasz	Wang, Qing
Hagedorn, Bastian	Naumchev, Alexandr	Wang, Shuling
Haidl, Michael	Navas, Jorge A.	Wimmel, Harro
Hallett, Joseph	Oancea, Cosmin	Ziegler, Martin
Hujsa, Thomas	Rasch, Ari	



# Contents

An Architecture for Non-invasive Software Measurement. . . . .	1
<i>Vasilii Artemev, Vladimir Ivanov, Manuel Mazzara, Alan Rogers, Alberto Sillitti, Giancarlo Succi, and Eugene Zouev</i>	
A Human-in-the-Loop Perspective for Safety Assessment in Robotic Applications . . . . .	12
<i>Mehrnoosh Askarpour, Dino Mandrioli, Matteo Rossi, and Federico Vicentini</i>	
Multi-level Static Analysis for Finding Error Patterns and Defects in Source Code. . . . .	28
<i>Andrey Belevantsev and Arutyun Avetisyan</i>	
Pipelined Bottom-Up Evaluation of Datalog Programs: The Push Method . . .	43
<i>Stefan Brass and Heike Stephan</i>	
A Platform for Security Monitoring of Multi-cloud Applications. . . . .	59
<i>Pamela Carvalho, Ana R. Cavalli, and Wissam Mallouli</i>	
The Hybrid Multidimensional-Ontological Data Model Based on Metagraph Approach . . . . .	72
<i>Valeriy M. Chernenkiy, Yuriy E. Gapanyuk, Anatoly N. Nardid, Anton V. Gushcha, and Yuriy S. Fedorenko</i>	
PosDB: A Distributed Column-Store Engine. . . . .	88
<i>George Chernishev, Viacheslav Galaktionov, Valentin Grigorev, Evgeniy Klyuchikov, and Kirill Smirnov</i>	
Microservices: How To Make Your Application Scale. . . . .	95
<i>Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina</i>	
Static Binary Code Instrumentation for ARM Architecture . . . . .	105
<i>Mikhail Ermakov</i>	
A Behavioural Theory for Reflective Sequential Algorithms. . . . .	117
<i>Flavio Ferrarotti, Klaus-Dieter Schewe, and Loredana Tec</i>	
Lightweight Non-intrusive Virtual Machine Introspection. . . . .	132
<i>Natalia Fursova, Pavel Dovgalyuk, Ivan Vasiliev, and Vladimir Makarov</i>	

A Distributed Approach to Coreference Resolution in Multiagent Text Analysis for Ontology Population . . . . .	147
<i>Natalia Garanina, Elena Sidorova, and Irina Kononenko</i>	
A Framework for Dynamical Construction of Software Components . . . . .	163
<i>Efim Grinkrug</i>	
A Transformation-Based Approach to Developing High-Performance GPU Programs . . . . .	179
<i>Bastian Hagedorn, Michel Steuwer, and Sergei Gorlatch</i>	
Domain Engineering the Magnolia Way. . . . .	196
<i>Magne Haveraaen</i>	
Approximating Event System Abstractions by Covering Their States and Transitions . . . . .	211
<i>Jacques Julliand, Olga Kouchnarenko, Pierre-Alain Masson, and Guillaume Voiron</i>	
Implementing the Symbolic Method of Verification in the C-Light Project . . . . .	227
<i>Dmitry Kondratyev</i>	
Highlights of the Rice-Shapiro Theorem in Computable Topology . . . . .	241
<i>Margarita Korovina and Oleg Kudinov</i>	
A Memory Model for Deductively Verifying Linux Kernel Modules . . . . .	256
<i>Mikhail Mandrykin and Alexey Khoroshilov</i>	
Indexing of Hierarchically Organized Spatial-Temporal Data Using Dynamic Regular Octrees. . . . .	276
<i>Sergey Morozov, Vitaly Semenov, Oleg Tarlapan, and Vladislav Zolotov</i>	
An Approach to the Validation of XML Documents Based on the Model Driven Architecture and the Object Constraint Language . . . . .	291
<i>Denis A. Nikiforov, Dmitriy V. Korj, and Ruslan L. Sivakov</i>	
Compositional Relational Programming with Name Projection and Compositional Synthesis . . . . .	306
<i>Görkem Paçacı, Steve McKeever, and Andreas Hamfelt</i>	
WhaleProver: First-Order Intuitionistic Theorem Prover Based on the Inverse Method. . . . .	322
<i>Vladimir Pavlov and Vadim Pak</i>	
Distributed In Situ Processing of Big Raster Data in the Cloud. . . . .	337
<i>Ramon Antonio Rodrigues Zalipynis</i>	

Statistical Approach to Increase Source Code Completion Accuracy . . . . . 352  
*Valeriy Savchenko and Alexander Volkov*

Using the Subject Area Ontology for Automating Learning Processes  
and Scientific Investigation. . . . . 364  
*Dmitry Shachnev and Dmitry Karpenko*

Runtime Specialization of PostgreSQL Query Executor . . . . . 375  
*Eugene Sharygin, Ruben Buchatskiy, Roman Zhuykov, and Arseny Sher*

MicroTESK: A Tool for Constrained Random Test Program Generation  
for Microprocessors. . . . . 387  
*Alexander Kamkin and Andrei Tatarnikov*

Enriching Textual Xtext-DSLs with a Graphical GEF-Based Editor. . . . . 394  
*Marcel Toussaint and Thomas Baar*

Towards Automated Static Verification of GNU C Programs . . . . . 402  
*Evgeny Novikov and Ilja Zakharov*

Domain Specific Semantic Validation of Schema.org Annotations. . . . . 417  
*Umutcan Şimşek, Elias Kärle, Omar Holzknacht, and Dieter Fensel*

**Author Index** . . . . . 431

# An Architecture for Non-invasive Software Measurement

Vasilii Artemev, Vladimir Ivanov<sup>(✉)</sup>, Manuel Mazzara, Alan Rogers,  
Alberto Sillitti, Giancarlo Succi, and Eugene Zouev

Innopolis University, Innopolis, Russian Federation  
vasart@gmail.com,  
{v.ivanov,m.mazzara,a.rogers,a.sillitti,g.succi,e.zuev}@innopolis.ru

**Abstract.** Analysis of data related to software development helps to increase quality, control and predictability of software development processes and products. However, collecting such data is a complex task. A non-invasive collection of software metrics is one of the most promising approaches to solve the task. In this paper we present an approach which consists of four parts: collect the data, store all collected data, unify the stored data and analyze the data to provide insights to the user about software product or process. We employ the approach to the development of an architecture for non-invasive software measurement system and explain its advantages and limitations.

**Keywords:** Software metrics collection  
Non-invasive software measurement · Software architecture

## 1 Introduction

Analysis of data related to software development helps to increase quality, control and predictability of both a development process and a resulting software product [34]. Collecting such data gives an opportunity to reconstruct software development process and produce insights on how to improve it. However, collecting the data is a complex task [16, 25]. An option is always collect the data ex-post through questionnaires and qualitative (some times with a level of subjectivity) [33]. However, a non-invasive collection of software metrics is one of the most promising approaches to solve the task [8, 26, 35].

There are systems which are targeting the area, but new available technologies, frameworks, libraries and tools enable a novel architecture for non-invasive measurement and analysis of software. Existing systems for non-invasive data collection typically use two types of metrics: software product metrics and software process metrics [6]. The data about software products and software development processes could be collected from developers' machines, smartphones, smart things, product repositories, task and defect tracking tools. The variety of sources and possible tools for data collection as well as many possible scenarios for data analysis make an issue of architectural design for developers of non-invasive software measurement systems.

The main goal of this preliminary study is to establish basic approach and principles of system architecture for non-invasive software measurement systems. The contribution of the work is focused on three aspects: (i) an approach that guides design decisions; (ii) a set of core elements for such systems and (iii) an analysis of architectural decisions.

In Sect. 2 we present the system architecture for non-invasive software metrics collection. In Sect. 3 we discuss the architectural decisions made; and in Sect. 4 we demonstrate a use case of the system. Sections 5 and 6 are devoted to related works and conclusions.

## 2 An Architecture for Non-invasive Metrics Collection

In this section, we present the approach to collect and analyse data as well as the system architecture and the underlying technologies in use. Although systems for non-invasive data collection have been presented before (see Sect. 5 for a comprehensive account), the approach presented in this paper is peculiar of this specific work, and represents one the major contribution of the study.

**Collect-Store-Unify-Analyze (CSUA) approach.** This approach consists of four parts: first of all we *collect* the data, such as metrics and events, from numerous distributed heterogeneous data sources. Second, we *store* all collected data in raw format suitable for future use. The third part consists of data *unifiers*, which can extract different relational data representations from non-relational stored data. Finally, we *analyze* the data providing insights to the user about observed product or process.

The CSUA approach guides the design of the architecture of a system for non-invasive software metrics collection. The architecture developed according to the CSUA approach presented in Fig. 1. Basic purposes for such architectural design are collection, storage and analysis of metrics as well as flexible representation of data in dashboards. The core elements of the architecture are:

- *agents* for collecting data;
- *databases*: document-oriented and relational;
- data *unifiers* and data *exporters*;
- *dashboards* and applications for data analysis.

The following subsections describe these components and major data flows in the system.

**Agents.** A system for non-invasive software metrics collection gathers data about software products and software development processes. The data sources usually include developers’ machines, smartphones and other devices; product repositories, task and defect tracking tools used in collaborative development. Data collection can be performed by multiple software agents of various types and kinds. The main purpose of an agent is data collection about a product and/or a process. There are multiple levels for agents to operate and collect data (e.g. OS-level agents, browser-level agents, IDE-level agents, etc.).

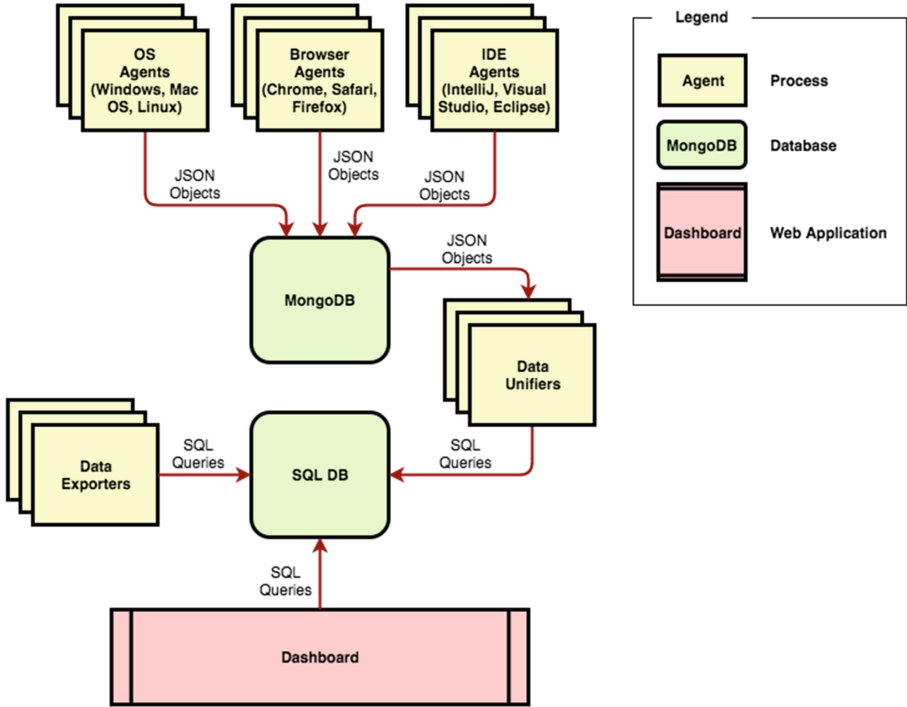


Fig. 1. Data flow in the system

OS agents are background operating system services for Windows, Linux, and Mac OS. Browser agents are extensions for Chrome, Firefox, and Safari. IDE agents are collecting data from Visual Studio, IntelliJ IDEA, Eclipse, XCode. The system is not limited only to these types, we are planning to add agents for bug tracking systems, version control systems, etc. Agents are in an early development phase at the moment<sup>1</sup>. Moreover, data could be normalized in many different ways, but we do not want to force one common data schema to every agent, we will make this decision later (according to Lean Development principles [20] in the “Data Unifier” component).

**Document-oriented database.** To store data collected by agents we use document-oriented database – MongoDB. The reasons why we chose this database is that it provides easy sharding of data, horizontal scaling and it uses JSON documents to store data.

A connector between an agent and a document-oriented database works in the following manner. An agent pushes data to a common document-oriented database over HTTP channel using RESTful API and JSON documents as data representation. We only impose a common high-level structure of JSON documents. A listing with example of a JSON document is provided below.

<sup>1</sup> User interface for one of prototype agents is shown in Fig. 2.

```

1  {
2  "timestamp": "2016-11-15T13:25:43.511Z",
3  "agent": {
4    "code_name": "MacOS developer's agent",
5    "full_name": "Developer's activity collector",
6    "secret_key": "6a81d622-5e24-4d9e-adc0-e3f7f2d93ac7",
7    "install_guid": "2187b011-6b9d-4d86-8083-dd09a0d73019"
8  },
9  "metrics": {
10   "event_id": "4a8acf6e7fbadc242de5b4f3",
11   "event_type": "web-browsing",
12   "event_duration": 1800,
13   "user": {
14     "username": "student",
15     "company": "Innopolis University"
16   },
17   "host": {
18     "host_name": "lab5_pc1",
19     "ip_address": "10.90.121.49",
20     "mac_address": "FF-FF-FF-FF-FF-FF",
21     "os_version": "macOS 10 Sierra Version 10.12.1",
22     "sw_version": "Safari Version 10.0.2 (12602.3.12.0.1)",
23   },
24   "sample_metric_data" : [
25     "stackoverflow.com", "google.com", "youtube.com"
26   ]
27 }
28 }

```

The document consists of three parts:

1. Timestamp
2. Agent information
3. Collected metrics

In the example, the top-level fields “timestamp” and “agent” describe the metadata, while the “metrics” part stores the actual data. The schema of the collected data may depend on an agent, but metadata fields stay the same across different agents. A sample user interface of an agent collecting process data is represented in Fig. 3. (Sect. 4).

**Data unifiers.** Data unifiers are processes which transform a set of JSON documents into rows and tables of a relational database. Resulting schema in each data unifier could be different depending on type of analysis that a customer may want to perform. Data unifiers pull data from MongoDB over HTTP channel using the same RESTful API as agents do.

**Relational database.** There could be multiple relational databases which our system may need to connect to. Hence, each data unifier serves as an adapter that write data to its own database.

**Data exporters.** The architecture provides data exporter component, so users of the system could do their own analysis. Basically, data exporters convert data to several well-recognized formats, like csv-file, arff-file, etc.

**Dashboarding applications.** Dashboard is an application which supports decision making by simplifying the data and representing it in a visual form. Backend part of a dashboarding application connects to a relational database. Frontend

is rich with graphs, charts, and data visualization. A developer of dashboarding applications may want more details later, so our system should be ready to adapt to these changes. That’s why modifiability of the system is highly demanded feature.

### 3 Discussion of Architectural Decisions

In this section we discuss significant architectural decisions, what options we considered and why we chose the structures that have been presented above. These architectural decisions affect attributes of the system, therefore we discuss them together in Table 1.

Attributes such as extensibility, modifiability and consistency would benefit from a migration into the microservice paradigm [5]. Recent projects of our team demonstrated an effective use and deploy of the paradigm in the field of ambient intelligence and smart buildings [22,23], in particular when associated with programming languages specifically designed with this purpose [1,2], and with adequate programming abstractions [21,32].

**Table 1.** Architectural decisions and motivation behind them

Attribute name	Arguments
Extensibility	Proposed architecture allows to add new agents and new analysis tools without downtime or reconfiguration
Security and Privacy	The system could be deployed in multiple organizations So we need to provide reasonable authorization, roles and access restriction settings
Performance	We need high-performance on write. There could be thousands of agents trying to write their data into document-oriented database at the same time
Consistency	We do not require strong consistency, eventual consistency should be fine
Modifiability	We require high modifiability of database schema
Scalability	We need horizontal scalability in terms of volume of data

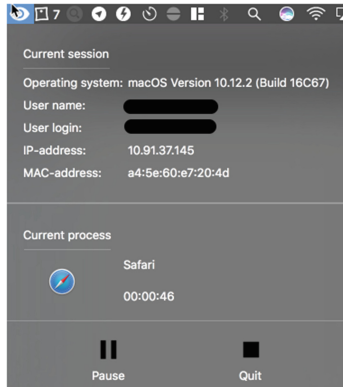
### 4 Use Case: A MacOS Agent Prototype

In this section, we show a common use case of the CSUA approach. We demonstrate such approach by the MacOS collector prototype. At the moment, only a prototype client-side application has been developed; it collects and transfers data for storage into the server (Figs. 2 and 3).

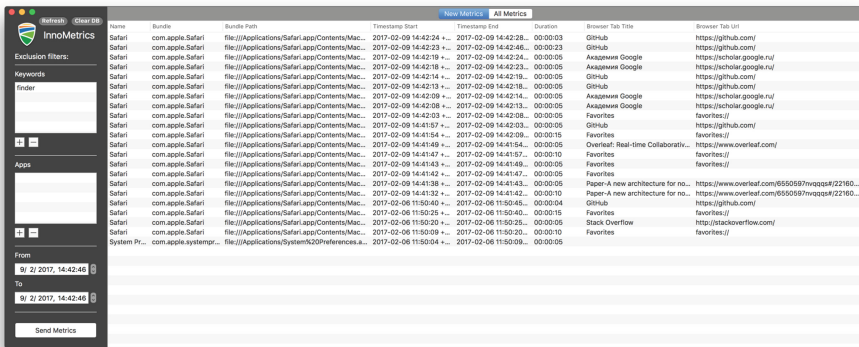
**Step 1: Collecting data.** A MacOS agent collects data in background and can be stopped at any time (see Fig.2). At any time, the user may review the collected data, apply a filter to collected records and submit them. This



possibility to manually stop, review and filter data before a submission makes the application friendly to users (especially to those users, who may consider it a spyware).



**Fig. 2.** User interface of a MacOS agent collecting data about user activity.



**Fig. 3.** User interface of a MacOS agent that represents collected data and transfers data to the server.

**Step 2: Store, filter and transfer data.** The interface for data transfer has several useful functions for accessing a collected dataset. A user may switch between newly collected (and not yet submitted) records and historical (submitted) records. In addition, there are three types of filters: a keyword filter, a filter by application and date/time filter (Fig. 3).

## 5 Related Works on Architectures for Non-invasive Measurement Systems

Over the past ten years several non-invasive measurement systems have been developed. In this section we review the following systems with emphasis on architecture:

- PRO Metrics (PROM);
- ElectroCodeoGram (ECG);
- Empirical Project Monitor (EPM);
- Hackystat.

### 5.1 PRO Metrics

PRO Metrics [10,27,28,30] is a distributed architecture for collecting software metrics and Personal Software Process (PSP) data [14]. PROM is based on Service-Oriented Programming development technique [31]. A client application stores collected information in XML file and does not deal with data transfer. This decision makes client-side components simpler. A transfer tool is separate client-application that transfers collected data and provides user authentication. Server-side components need to be installed and maintained only on one machine, therefore the overall complexity of the system is low. But in case of installation client components on many machines with different environment it becomes not a trivial task for a system administrator.

### 5.2 ElectroCodeoGram

ElectroCodeoGram is a modular framework [24] aimed at micro-process research and discovering patterns in the sequence of events which describe the same programming behavior. For instance (i) copy and paste some piece of code with desired functionality and (ii) refactor code and make a function with needed parameters, represent two different patterns (or episodes) solving the same task. ECG supports micro-process research. It automatically records micro-process data using ECG Sensors; sends data to the central collection and analysis system. Data is transported over network sockets or SOAP.

### 5.3 Empirical Project Monitor

Empirical Project Monitor [18,19] is a system that automatically collects data (by “pulling”) from three different repositories:

- Configuration management systems;
- Mailing list managers (e.g. Mailman, Majordomo);
- Issue tracking systems (e.g. Bugzilla).

The EPM system consist of three components:

- Automatic data collection. EPM automatically collects data from repositories.
- Format translation and data store. EPM converts collected data to XML format. Converted data is stored in the PostgreSQL database.
- Analysis and visualization. EPM gets data for analysis from the database for visualization.

## 5.4 Hackystat

Hackystat [11–13] is a system for automatic collecting development metrics from sensors (attached to development tools). Hackystat sends data to the server where this data is analyzed. Its sensors are able to collect:

- activity data (e.g. which file is under modification of developer);
- size data (e.g. lines of code);
- defect data (e.g. number of pass/fail status of unit tests).

A developer should install one or more sensors to begin using Hackystat and then register with its server. In later versions of Hackystat its architecture has been criticized for growing complexity; developers made a decision to review the architecture and reimplement Hackystat in a service-oriented architecture (SOA). The main challenges for this revision were almost complete reimplementation of the system and the need for system developers to move to new architectural concept and libraries.

## 6 Conclusion and Future Work

Non-invasive collection of software metrics demonstrated to be effective in the field of software measurement [3, 7], including also non traditional situations [9]. Several systems for non-invasive data collection have been presented in the past, also for mobile contexts [4, 29]. However, the approach presented in this paper is innovative in its own nature: for the peculiarity of the data flow and for the specific architecture adopted, as well as for the underlying architectural decisions. The architecture is designed to provide an high level of Scalability and Modifiability, as well as a direct way to extend the system with new types of agents, and, when possible, uses open source components [15]. Forthcoming steps include development of agents for operating systems (Windows and Linux), specific IDEs, popular browsers, version tracking systems, task tracking systems, and defect tracking systems.

Recent trends and development in the field of software architecture has shown an increasing attention towards the microservice architecture, which promises to help managing scalability, elasticity and robustness [5]. It is under consideration the possibility to migrate from the current design to this new approach. At the moment, there is no concrete work in this direction in the field of non-invasive collection, therefore it would represent an innovative trait of the system.

Social networks have also seen an emerging interest in data collection, for example for real-time trust measurement [17]. In this context, the possibility to aggregate data from the network and local users data (for example, on machine or web usage) may represent an effective synergy that requires further investigation.

## References

1. Bandura, A., Kurilenko, N., Mazzara, M., Rivera, V., Safina, L., Tchitchigin, A.: Jolie community on the rise. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), pp. 40–43 (2016)
2. Guidi, C., Lanese, I., Mazzara, M., Montesi, F.: Microservices: a language-based approach. In: Mazzara, M., Meyer, B. (eds.) *Present and Ulterior Software Engineering*, pp. 217–225. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-67425-4\\_13](https://doi.org/10.1007/978-3-319-67425-4_13)
3. Coman, I.D., Sillitti, A., Succi, G.: Investigating the usefulness of pair-programming in a mature agile team. In: Abrahamsson, P., Baskerville, R., Conboy, K., Fitzgerald, B., Morgan, L., Wang, X. (eds.) *XP 2008. LNBP*, vol. 9, pp. 127–136. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-68255-4\\_13](https://doi.org/10.1007/978-3-540-68255-4_13)
4. Corral, L., Sillitti, A., Succi, G., Garibbo, A., Ramella, P.: Evolution of mobile software development from platform-specific to web-based multiplatform paradigm. In: *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 181–183. ACM (2011)
5. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. In: Mazzara, M., Meyer, M. (eds.) *Present and Ulterior Software Engineering*, pp. 195–216. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12)
6. Fenton, N.E., Pfleeger, S.L.: *Software Metrics: A Rigorous and Practical Approach*, 2nd edn. PWS Publishing Co., Boston (1998)
7. Fronza, I., Sillitti, A., Succi, G.: An interpretation of the results of the analysis of pair programming during novices integration in a team. In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 225–235. IEEE Computer Society (2009)
8. Janes, A., Scotto, M., Sillitti, A., Succi, G.: A perspective on non invasive software management. In: *Instrumentation and Measurement Technology Conference (IMTC)* (2006)
9. Janes, A.A., Succi, G.: The dark side of agile software development. In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 215–228. ACM (2012)
10. Jermakovics, A., Sillitti, A., Succi, G.: Mining and visualizing developer networks from version control systems. In: *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 24–31. ACM (2011)
11. Johnson, P.M.: Requirement and design trade-offs in hackystat: an in-process software engineering measurement and analysis system. In: *ESEM*, vol. 7, pp. 81–90 (2007)
12. Johnson, P.M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S., Doane, W.E.J.: Beyond the personal software process: metrics collection and analysis for the differently disciplined. In: *Proceedings of the 25th International Conference on Software Engineering*, pp. 641–646. IEEE Computer Society (2003)

13. Johnson, P.M., Kou, H., Agustin, J.M., Zhang, Q., Kagawa, A., Yamashita, T.: Practical automated process and product metric collection and analysis in a classroom setting: lessons learned from hackystat-uh. In: Proceedings of the 2004 International Symposium on Empirical Software Engineering, ISESE 2004, pp. 136–144. IEEE (2004)
14. Kivi, J., Haydon, D., Hayes, J., Schneider, R., Succi, G.: Extreme programming: a university team design experience. In: 2000 Canadian Conference on Electrical and Computer Engineering, vol. 2, pp. 816–820. IEEE (2000)
15. Kovács, G.L., Drozdik, S., Succi, G., Zuliani, P.: Open source software for the public administration. In: Proceedings of the 6th International Workshop on Computer Science and Information Technologies (2004)
16. Maurer, F., Succi, G., Holz, H., Kötting, B., Goldmann, S., Dellen, B.: Software process support over the internet. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 642–645. ACM, May 1999
17. Mazzara, M., Biselli, L., Greco, P.P., Dragoni, N., Marraffa, A., Qamar, N., de Nicola, S.: Social networks and collective intelligence: a return to the Agora. IGI Global (2013)
18. Ohira, M., Yokomori, R., Sakai, M., Matsumoto, K., Inoue, K., Barker, M., Torii, K.: Empirical project monitor: a system for managing software development projects in real time. In: International Symposium on Empirical Software Engineering, Redondo Beach, USA (2004)
19. Ohira, M., Yokomori, R., Sakai, M., Matsumoto, K., Inoue, K., Torii, K.: Empirical project monitor: a tool for mining multiple project data. In: International Workshop on Mining Software Repositories (MSR 2004), pp. 42–46. IET (2004)
20. Poppendieck, M., Poppendieck, T.D., Poppendieck, T.: Lean Software Development: An Agile Toolkit. The Agile Software Development Series. Addison-Wesley, Boston (2003)
21. Safina, L., Mazzara, M., Montesi, F., Rivera, V.: Data-driven workflows for microservices (genericity in Jolie). In: IEEE International Conference on Advanced Information Networking and Applications (2016)
22. Salikhov, D., Khanda, K., Gusmanov, K., Mazzara, M., Mavridis, N.: Jolie good buildings: Internet of Things for smart building infrastructure supporting concurrent apps utilizing distributed microservices. In: Proceedings of the 1st International Conference on Convergent Cognitive Information Technologies, pp. 48–53 (2016)
23. Salikhov, D., Khanda, K., Gusmanov, K., Mazzara, M., Mavridis, N.: Microservice-based IoT for smart buildings. In: Proceedings of the 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA) (2017)
24. Schlesinger, F., Jekutsch, S.: Electrocodeogram: an environment for studying programming. In: Workshop on Ethnographies of Code, Infolab21, pp. 30–31. Lancaster University, UK (2006)
25. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: Dealing with software metrics collection and analysis: a relational approach. *Stud. Inf. Univ.* **3**(3), 343–366 (2004)
26. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: Non-invasive product metrics collection: an architecture. In: Proceedings of the 2004 Workshop on Quantitative Techniques for Software Agile Process, QUTE-SWAP 2004, pp. 76–78. ACM, New York (2004)
27. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: A non-invasive approach to product metrics collection. *J. Syst. Archit.* **52**(11), 668–675 (2006)

28. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Collecting, integrating and analyzing software metrics and personal software process data. In: *EUROMICRO*, vol. 3, p. 336 (2003)
29. Sillitti, A., Janes, A., Succi, G., Vernazza, T.: Measures for mobile users: an architecture. *J. Syst. Archit.* **50**(7), 393–405 (2004)
30. Sillitti, A., Succi, G., De Panfilis, S.: Managing non-invasive measurement tools. *J. Syst. Archit.* **52**(11), 676–683 (2006)
31. Sillitti, A., Vernazza, T., Succi, G.: Service oriented programming: a new paradigm of software reuse. In: Gacek, C. (ed.) *ICSR-7. LNCS*, vol. 2319, pp. 269–280. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-46020-9\\_19](https://doi.org/10.1007/3-540-46020-9_19)
32. Tchitchigin, A., Safina, L., Mazzara, M., Elwakil, M., Montesi, F., Rivera, V.: Refinement types in Jolie. In: *Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE* (2016)
33. Tumyrkin, R., Mazzara, M., Kassab, M., Succi, G., Lee, J.Y.: Quality attributes in practice: contemporary data. In: Jezic, G., Chen-Burger, Y.-H.J., Howlett, R.J., Jain, L.C. (eds.) *Agent and Multi-Agent Systems: Technology and Applications. SIST*, vol. 58, pp. 281–290. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-39883-9\\_23](https://doi.org/10.1007/978-3-319-39883-9_23)
34. Vera-Baquero, A., Colomo-Palacios, R., Molloy, O.: Business process analytics using a big data approach. *IT Prof.* **15**(6), 29–35 (2013)
35. Vernazza, T., Granatella, G., Succi, G., Benedicenti, L., Mintchev, M.: Defining metrics for software components. In: *5th World Multi-Conference on Systemics, Cybernetics and Informatics, Florida*, vol. 11, pp. 16–23 (2000)

# A Human-in-the-Loop Perspective for Safety Assessment in Robotic Applications

Mehrnoosh Askarpour<sup>1</sup>(✉), Dino Mandrioli<sup>1</sup>, Matteo Rossi<sup>1</sup>,  
and Federico Vicentini<sup>2</sup>

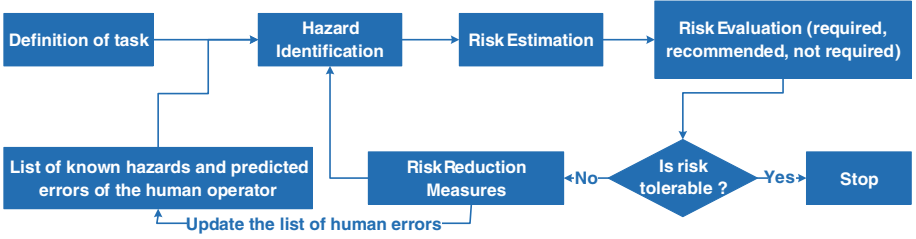
<sup>1</sup> DEIB, Politecnico di Milano, Milan, Italy  
{mehrnoosh.askarpour,dino.mandrioli,matteo.rossi}@polimi.it  
<sup>2</sup> CNR, ITIA, Milan, Italy  
federico.vicentini@cnr.itia.it

**Abstract.** Human-Robot Collaborative (HRC) applications pose new challenges in the assessment of their safety, due to the close interaction between robots and human operators. This entails that a human-in-the-loop perspective must be taken, at both the design and the operation level, when assessing the safety of these applications. In this paper we present an extension of a tool-supported methodology compatible with current ISO 10218-2 standard, called SAFER-HRC, which: (i) takes into account the possible behaviors of human operators—such as mistakes and misuses while working with the robot (operational level)—and (ii) exploits the expertise of safety engineers in order to incrementally update and adjust the model of the system (design level). The methodology is supported by a tool that allows designers to formally verify the modeled HRC applications in search of safety violations in an iterative manner.

**Keywords:** Safety analysis · Formal verification · Safety rules  
Human-Robot Collaboration · Human in the loop

## 1 Introduction

In Human-Robot Collaborative (HRC) applications, workers and machines cooperate in close proximity in a common work-cell, sometimes with direct physical contacts, either voluntary or accidental. As such, the design of an application may lead to hazardous situations for the operator working in the work-cell, either due to operations (*e.g.*, tools, motions, etc.) or to the behavior of the operator, which is inherently non-deterministic; for example, the possibility of operator's deviations from the execution instructions cannot entirely be ruled out. Further, in collaborative applications, the operator has alternatives in performing the application (*i.e.*, human flexibility), potentially leading to unforeseeable combinations of tasks by humans and robots that can generate hazards that were not initially foreseen by designers of the system. The unforeseen (*e.g.*, errors) and unwanted (*e.g.*, intentional misuses) behaviors by the operator make it more difficult to guarantee human safety in HRC applications than in those executed



**Fig. 1.** The refinement of traditional risk assessment technique with a list of hazards categorized in ISO 10218 and reasonably foreseen human errors.

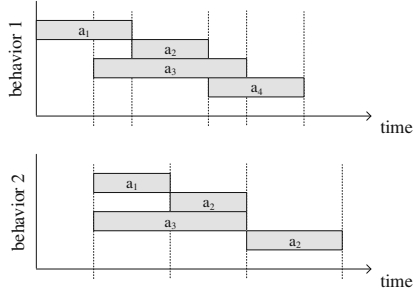
within fenced areas or segregated robotic cells; this highlights the need for a human-centric approach to the risk analysis of such applications. We aim to obtain a reliable analysis by defining a methodology that adopts a human-in-the-loop perspective at both the operational and the design level, in which the behavior and predictable errors of a human operator are captured in a suitably formalized module. This module will be a part of a larger formal model of the collaborative system, which describes activities and decisions of the operator at operation time, and hence takes them into account when verifying the safety of the system. The methodology should be able to identify hazards in an automated manner so no hazardous situation is left unconsidered, and yet rely on the expertise and judgment of robotic safety engineers and enable them to (i) monitor the results of the analysis, (ii) study multiple suggestions provided by the methodology to remedy each risky situation, and (iii) choose the most suitable safety measure for the analyzed system.

The classical risk assessment approach, depicted in Fig. 1, consists of four phases repeated iteratively:

- **Hazard identification**, where presence of hazards is determined.
- **Risk Estimation**, where the consequences of identified hazards (*e.g.*, severity of injuries) are estimated and a quantized risk value is computed.
- **Risk Evaluation**, where the risk value is compared with a target level (*e.g.*, negligible or acceptably low).
- Introduction of **Risk Reduction Measures** (RRM), where proper design or control strategies are set in place for avoiding or mitigating consequences of non-negligible hazards.

In an earlier work [2], we introduced a methodology behind the framework we are building called SAFER-HRC (Safety Analysis through Formal vERification in Human-Robot Collaboration). It uses concepts of temporal logic and satisfiability checking to automate as much as possible the classic risk assessment approach highlighted above. It is rooted in the idea of building a formal model of a collaborative system using temporal logics—adopting a discrete notion of time with temporal domain  $\mathbb{N}$ —and then formally verifying desired safety properties specified according to standard ISO 10218-2 [14]. The model focuses on





**Fig. 2.** Example of two possible behaviors of the model regarding the sequence of actions.  $a_2$  can start its execution only when  $a_1$  is complete.  $a_4$  also needs to wait for the completion of  $a_2$ , whereas  $a_3$  can execute in parallel with any other action. Depending on when any of the actions are ready to execute, their overall order over time may change.

two main aspects. The first one concerns capturing the tasks that should be executed within the application as a set of smallest possible functional units—*i.e.*, *elementary actions*—and the identification of the safety requirements of each of them. The second concerns highlighting the role of the operator behavior (*e.g.*, mistakes and decisions about execution) in creating hazardous situations.

Our proposed environment provides pre-defined packages of formulae and application-specific information (*e.g.*, specification of the physical environment), that makes it easier for safety analyzers to model and verify HRC applications.

To define the formal model of the system we use a decidable fragment of the TRIO metric temporal logic [10]; the verification phase is supported by the Zot tool [1], an efficient bounded satisfiability checker for logic formulae [3], which exhaustively explores the state-space of traces of the model within a bounded time interval, to identify possible hazardous situations that designers have left out or forgot to consider in the design. The model is then incrementally updated to cover the identified situations and it is iteratively verified against desired safety properties until all of them are satisfied.

In this work, we extend the SAFER-HRC modeling approach to include: (i) activities that are executed concurrently by the operator and the robot; (ii) the nondeterminism of the operator behavior; and (iii) risk estimation in the form of methods consistent with international standards in the domain of safety of machinery [15]. The verification mechanism at the heart of SAFER-HRC exhaustively searches all different possible execution traces of the model, and in particular all alternative orderings of execution of the actions within a task that achieve the goal. For example, assume that both traces shown in Fig. 2 achieve the same goal, but starting  $a_4$  before completing  $a_3$  causes a hazard that does not occur otherwise (*e.g.*, a change in positioning of the operator w.r.t the robot). SAFER-HRC explores both traces, detects this hazard and computes its risk. If the resulting risk is non-negligible, the analyzer can try different modifications, compare their residual risks, choose the most efficient one and permanently add it to the model.

## 1.1 Related Work

Safety analysis techniques can be grouped in three categories: informal (*e.g.*, [13, 17]), semi-formal (*e.g.*, [11, 12, 18–20]) and formal. Formal techniques are fully mathematical solutions which lend themselves to analysis through automated verification tools. FMEA [4], FTA [21] and Markov techniques [23], are traditional examples of formal solutions [7], which are however not well-suited for HRC applications, as they cannot deal with unpredictable human interactions with robots. Model-based formal verification recently became popular in robotics, although to the best of our knowledge it has never been used directly for safety analysis of HRC applications. For example, [22] focuses on trustworthiness of robots and neglects the impact of human behavior on safety and performance. [26, 27] model a service robotic system (an assistant robot for disabled people) by the Brahms language [24] and verify it against safety requirements. [8, 25] also employ formal verification to analyze the safety of assistant robots.

Nevertheless, these works do not consider situations raised by cooperation of human and robot, neither they encompass the verification of systems against significant mechanical hazards which may hurt the human body. They mostly focus on assistant rather than collaborative industrial robots, so during the verification and hazard identification process they do not study human activities and interactions with robots. Thus, there is no reference to international standard ISO/TS 15066 [16]—complementing the widely adopted ISO 10218-2.

Further, the approaches described in [5, 6, 9] have a stronger focus on human-device interaction through interfaces—without physical involvement—thanks to their use of cognitive architectures. However, they usually require large manual customizations and have little reference to human fallibility and plausible errors.

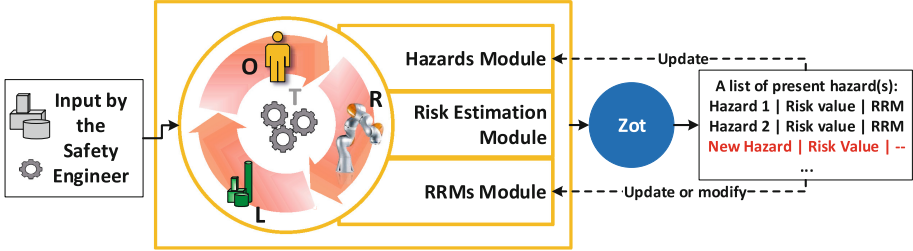
Conversely, this paper extends our formal approach SAFER-HRC by taking into account the operator behavior as a relevant factor in the system model. In this approach interaction with safety analyzers—who usually have a mechanical engineering background—is central, as their operational perspective helps to foresee operators’ errors and misuses during interaction with robots.

The rest of this paper is organized as follows: Sect. 2 explains how SAFER-HRC helps safety analyzers to create formal models and presents their semantics; Sect. 3 describes the encapsulation of the classic risk assessment approach within SAFER-HRC; Sect. 4 illustrates the application of SAFER-HRC to a real HRC application and Sect. 5 concludes.

## 2 Semantic Model

The SAFER-HRC approach depicted in Fig. 3 comprises four modules, of which the first one is explained in [2] and the other three are presented in this paper as extensions to the approach. They formalize the main aspects of an HRC application through the TRIO language. The *ORL-module* contains formal descriptions for operator  $O$ , robot  $R$  and layout  $L$ . The descriptions of  $O$  and  $R$  are generic and there can be multiple instances of each of them, according to the case study.

For example if there are two operators required in a system, then the safety analyzer provides this information as input to SAFER-HRC and two instances of  $O$  are created. It also includes a set of rules ( $T$ ) concerning the definition of tasks, which are application-independent and can be tailored for different cases. Some information is specific to each task and to the physical layout, hence the safety analyzer should provide it as input.



**Fig. 3.** Overview of SAFER-HRC methodology. The layout description, task to be performed and number of operator and robot instances are provided as input; the Hazards Module contains known significant hazards according to ISO10218 and also foreseen human errors; the Risk Estimation Module is compatible with ISO 14121.

A second module formalizes all the *significant hazards* that have been non-exhaustively defined and categorized in standards like [14], encapsulating established experience. It includes reasonably foreseeable errors that an operator can make, which can lead to hazardous situations. Hazard sources clearly need to be updated as soon as new situations and harms are experienced.

The *Risk estimation module* contains the information regarding the computation of a risk value for each detected hazard according to the hybrid method approach reported in ISO 14121 [15].

Finally, the *RRM module* defines all the risk reduction measures which are triggered by detection of non-negligible risk values.

In practice, these modules are instantiated for each specific HRC application by a joint team of mechanical engineers with experience in HRC applications and computer scientists with proficiency in formal verification activities. The rest of this section illustrates the contents of each module with more detail.

## 2.1 *ORL*-Module Formalizing Operator, Robot and Layout

This section elaborates on the details of three sets of TRIO formulae defined in the *ORL*-module. Set  $L$  defines relevant regions in the work-cell based on the workspace and the kinematics of the robot, and contains formulae to define adjacency or separation of these regions. This information is application-dependent and is provided by a safety analyzer in the form of inputs about geometry, kinematics and materials, for example through a questionnaire that gathers information as structured texts that can be converted into SAFER-HRC models.

Set  $O$  abstracts the new specification of ISO15066, in which experimental data is used to identify thresholds for acceptable pressure or force for different parts of the human body. The set models the human body with eleven most relevant parts and their limit values; it includes constraints to avoid unrealistic body shapes in the analysis (*e.g.*,  $\text{Alw}(p_{\text{head}} = p_{\text{arms}} \vee \text{adj}(p_{\text{head}}, p_{\text{arms}}))$ ) states that arms and head can not be at opposite ends of the work-cell and can only be in the same or adjacent layout regions at the same time).

Set  $R$  defines multiple variables to capture the position of the various parts of the robot in the layout, such as for example the position of robot links and end-effector. As for set  $O$ , set  $R$  also includes constraints on the structure of the robot. For example, let us consider the structure of a KUKA lightweight manipulator robot ([www.kuka-robotics.com](http://www.kuka-robotics.com)), which is available in our experimental setup, and which includes two links ( $R_1, R_2$ ), three main joints and an end-effector ( $EE$ ). The following formula indicates that  $R_1$ , which describes the position of the arm link, and  $R_2$ , which captures the forearm link, are always in the same or adjacent layout region(s):  $\text{Alw}(p_{R_1} = p_{R_2} \vee \text{adj}(p_{R_1}, p_{R_2}))$ .

## 2.2 Formalization of Tasks

The *ORL*-module includes also a set of formulae, called  $T$ , which describes tasks by breaking them down into elementary actions with defined execution chains to complete the task. In the following we present a new version of the set  $T$  of formulae, which has been deeply revised w.r.t the one initially introduced in [2].

**Actions.** The formal definition of an action  $a_i$  comprises three groups of formulae: *Pre-Conditions* ( $\text{pre}C_i$ ), *Safety-Constraints* ( $\text{sft}C_i$ ) and *Post-Conditions* ( $\text{pos}C_i$ ).

$\text{pre}C_i$  (resp.,  $\text{pos}C_i$ ) hold right before the start (resp., right after the termination) of the execution of  $a_i$ .  $a_i$  starts to execute only if all its pre-conditions are true, and when it terminates all of its post-conditions hold. For example, one of the pre-conditions for the action “Robot screwdrives the prepared fixtures on the work-piece” is that the robot should be positioned in the right spot for screwdriving. Thus, screwdriving will not start until the robot is in the right position. A post-condition for the same action is “the work-piece is completely fixed on position y”, so this action terminates only when this condition is true. Each  $a_i$  is also associated with some formulae  $\text{sft}C_i$ , which should be true while it is executing, otherwise the execution is paused until all  $\text{sft}C_i$  hold again. A safety constraint in the previous example is “the operator should keep holding the workpiece until the robot finishes screwdriving and fixing it” which should be true while the execution of the action is ongoing. Formulae  $\text{pre}C$  and  $\text{sft}C$  of actions are consistent with any specific order of execution of actions which is required in order to terminate the task. For example, if  $a_y$  should strictly execute before  $a_x$ , then it is stated as a pre-condition for  $a_y$  that  $a_x$  is terminated. However, other actions can be executed in between these two, and they do not necessarily execute after one another. Additionally, the model

of each action  $a_i$  includes: constant  $exeT_i \in \mathbb{N}$ , which captures its execution time;  $pmr_i \in \{\text{ro}, \text{op}\}$ , which declares if it should be performed by the robot or by the operator;  $sts_i \in \{\text{ns}, \text{wt}, \text{exe}, \text{ps}, \text{dn}\}$ , which captures its the state, and whose value changes over time according to the following rules i-v (note that all formulae below are implicitly quantified over time through operator  $\text{Alw}$ ).

- (i) An action remains **not started** (ns) until all of its pre-conditions hold.
- (ii) When all the pre-conditions of an action hold, it becomes **waiting** (wt).

$$sts_i = \text{wt} \wedge \text{Past}(sts_i = \text{ns}, 1) \Rightarrow \text{Past}(\text{pre}C_i, 1)$$

The action remains so until its safety constraints start holding and thus becomes executing (exe). If pre-conditions stop holding before the satisfaction of safety constraints, the action becomes ns.

- (iii) Safety constraints must hold during the execution:  $sts_i = \text{exe} \Rightarrow \text{sft}C_i$ .
- (iv) An executing action becomes **paused** (ps) as its safety constraints are violated, and remains so until the violations are resolved.

$$sts_i = \text{exe} \wedge \text{Futr}(\neg \text{sft}C_i, 1) \Rightarrow \text{Futr}(sts_i = \text{ps} \wedge \text{Until}_w(sts_i = \text{ps}, \text{sft}C_i), 1)$$

- (v) An Action becomes **done** when its post-conditions are satisfied and will remain so for the rest of the execution of the task.

$$sts_i = \text{exe} \wedge \text{Futr}(\text{pos}C_i, 1) \Rightarrow \text{Futr}(sts_i = \text{dn} \wedge \text{AlwF}(sts_i = \text{dn}), 1)$$

- (vi) If a task requires to repeat action  $a_k$  (e.g., within a loop iterated a fixed number  $n$  of times), multiple separate actions which are instances of  $a_k$  are defined, one for each repetition (e.g.,  $a_k^1, \dots, a_k^n$ ). For each iteration, the actions of previous iterations must be complete ( $\forall i : sts_k^{(i-1)} = \text{dn} \in \text{pre}C_k^i$ ).

**Modeling the Execution of Actions.** Actions are divided into two groups according to their performer. The **actions done by the robot** are considered fully deterministic, meaning that the following rules hold for them:

- (i) Unlike operator actions which need human's act (as explained below), robot's actions start execution deterministically at the instant immediately following the one when the state is waiting and the safety constraints are satisfied.  
 $pmr_i = \text{ro} \wedge sts_i = \text{wt} \wedge \text{sft}C_i \Rightarrow \text{Futr}(sts_i = \text{exe}, 1)$ .
- (ii) The execution order of actions is implicit in their pre/post-conditions and safety constraints. For example, screwdriving ( $a_z$ ) should execute after robot moves to the right place ( $a_x$ ) and the operator brings a workpiece to the screwdriving spot ( $a_y$ ); hence, the termination of the two latter actions is part of the pre-conditions of screwdriving:  $\text{pre}C_z \Rightarrow sts_x = \text{dn} \wedge sts_y = \text{dn}$ .
- (iii) Multiple actions execute concurrently only when it is explicitly mentioned within their definitions. For example, when robot grabs a workpiece and carries it to another position in the work-cell, then gripping ( $a_g$ ) and moving to target position ( $a_m$ ) must concurrently execute:  $\text{sft}C_m \Rightarrow sts_g = \text{exe}$ .

However, **actions done by the operator are non-deterministic**. Pre-conditions of an action may hold, but the operator may not execute it (*e.g.*, if he is absent-minded or distracted). Still, we assume that the operator starts executing the action within a finite time  $\Delta$ , so that task termination is guaranteed:

- (i) An operator’s waiting action becomes either *exe* or *ns* within  $\Delta$  time units.

$$pmr_i = op \wedge sts_i = wt \Rightarrow \text{WithinF}(sts_i = exe \vee sts_i = ns, \Delta)$$

- (ii) The operator can not execute more than two actions at the same time. If several operator actions are waiting, nondeterministically, and not necessarily simultaneously, at most two of them will start executing within  $\Delta$  time units.

$$\neg \exists i, j, k (i < j < k \wedge \bigwedge_{z \in \{i, j, k\}} sts_z = exe \wedge pmr_z = op)$$

If a robot and an operator action have to execute concurrently, then this should be stated within their definition. For example, screwdriving can execute only when the operator is holding the workpiece, otherwise it might fall off, interrupting the action. So, it is mentioned in the pre-conditions and safety constraints of screwdriving that workpiece must be held. Notice that this does not mean that these two actions start simultaneously: holding by the operator ( $a_u$ ) can start earlier than screwdriving, but the key requirement is that it has to be executing along screwdriving:  $sftC_z \Rightarrow sts_u = exe$ .

### 2.3 Hazard Definition Module

This module contains the formal definition of hazards. For example the following formula defines a hazardous situation in which the end-effector of the robot—with a screwdriver mounted on it—is moving and close to the operator’s head. This situation, called  $hzd_x$ , could cause facial injury (*e.g.*, loss of an eye):

$$hzd_x \Leftrightarrow p_{head} = p_{EE} \wedge \neg(mode_{robot} = idle) \wedge EE_{type} = screwdriver$$

This module is updated if, during the iterative analysis, safety violations are detected which do not correspond to any defined hazard. It means that some hazardous situations have been overlooked in the initial model.

### 2.4 Risk Estimator Module

For each detected hazard a quantitative risk value is computed, to be later used as a criterion for selection of a suitable RRM.

ISO14121 indicates three possible values for the risk associated with each hazard:  $\forall hzd_i : risk_i \in \{1, 2, 3\}$ . For example, if  $risk_i = 2$  holds, then a strong RRM such as requiring the full stop of robot is necessary; if  $risk_i = 1$ , then a weaker RRM like pausing the robot for  $\alpha$  time instants and then continue could be enough and its application is only recommended and not necessary. Trivially, if  $risk_i = 0$  holds, then no RRM is required.

**Table 1.** Hybrid risk estimation technique according to ISO 14121.

		Class e(CI = Fr+Pr+Av)					Frequency (Fr)	Probability (Pr)	Avoid-ability (Av)			
		3-4	5-7	8-10	11-13	14-15						
Severity (Se)	4 irreversible injury						T < 1h	5	very high	5	impossible	5
	3 permanent injury						1h < T ≤ 24h	5	likely	4	possible	3
	2 reversible injury <sup>1</sup>						24h < T ≤ 2w	4	possible	3	likely	1
	1 reversible injury <sup>2</sup>						2w < T ≤ 1y	3	rarely	2		
							1y < T	2	negligible	1		
note	<sup>1</sup> medical attention <sup>2</sup> first aid	black = 2 (RRM required) gray = 1 (RRM recommended) white = 0 (RRM not required)					T exposure range					

To compute the risk value according to ISO 14121, four main parameters are required: the severity  $Se$  of the possible injury caused by the hazard, and the avoidability  $Av$ , frequency  $Fr$  and probability  $Pr$  of each hazard (notice that, in this context, probability is described through a rank from 1 to 5, going from “negligible” to “very highly probable”).

The value of  $Se$  depends on the involved part of the human body, on the type of hazard (*e.g.*, hit, entrapment), and on the robots’ force and speed; however, in the current formalization we only take into account body part and type of hazard, since the model does not include information about the physical dynamics of the robot. The other three parameters are initialized by empirical information and are statically embedded in the model. A class identifier  $CI$ , which is computed by  $Fr + Pr + Av$ , is combined with  $Se$  and is mapped to one of the possible values for risk according to a table such as Table 1. For example, with respect to this classification the severity of the hazard illustrated in Sect. 2.3 is equal to four ( $Se_x = 4$ ), and its risk value is computed through a formula such as  $hzd_x \wedge (14 \leq CI_x \leq 15) \rightarrow risk_x = 2$ .

## 2.5 RRM Module

For each hazard defined in the hazard module, one or more corresponding RRM is defined in the RRM module. When a hazard occurs in an execution trace of the system model, then SAFER-HRC reports the hazard’s presence, its related risk value and associated RRMs to the safety analyzer.

Each RRM can result in several changes in the system, hence also in the model. For example, an RRM might consist in modifying the precedence between actions, so as to avoid orderings in the execution of actions that are hazardous with a high risk; this would correspond to a change in the definitions of involved actions, such as adding constraints to their pre-conditions or safety constraints. RRMs can impact the conditions under which actions are executed, for example by reducing the speed of movement in action “robot moving to the bin”, or by pausing an action when certain conditions are not met; the introduction of such RRMs would result in a modification of the safety constraints of impacted actions, for example by adding suitable constraints. Another example of RRM

is a change in the layout design, such as covering sharp edges with pads, or leaving warning labels on equipments. Such physical modifications can impact the model in different ways; for example, covering sharp edges could result in a reduction of the risk value of a pinching hazard; the effect of a warning sign could be that the foreseeable behavior of the operator is modified, as one can assume that he will avoid certain situations. For example, an  $RRM_x$  for  $hzd_x$  in Sect. 2.3 is “stop any movement of the robot” and corresponds to the following definition:

$$RRM_x \Leftrightarrow p_{R_1} = \text{futr}(p_{R_1}, 1) \wedge p_{R_2} = \text{futr}(p_{R_2}, 1) \wedge p_{EE} = \text{futr}(p_{EE}, 1)$$

The RRM must remain true until the hazard is resolved, which is captured by the following formula:  $hzd_x \Rightarrow \text{Until}_w(RRM_x, \neg hzd_x)$ .

### 3 Overview of Iterative Risk Assessment

After the formal model of the target application is defined, it is verified against the following properties:

1. Whether any hazard is detected, and if it is, whether an effective RRM is triggered.
2. Whether there are hazardous situations which have yet to be identified in the hazard module, or combinations of known hazards that current RRMs do not mitigate.
3. Given the incremental nature of the methodology, whether the hazardous situations in the previous iterations are eliminated or remedied by the RRMs.

To address the first issue, the formal model of the application is evaluated against the property  $\text{Alw}(\forall hzd_x(\text{risk}_x = 0))$ . If the property holds, no hazard with non-negligible risk value occurs in the application. Otherwise, the verification tool returns a system trace which highlights, thanks to the definitions introduced in Sect. 2.3, the occurrences of hazards that need to be mitigated.

The second issue is addressed by checking whether there are possible system executions in which an action is paused (due to a violation of its safety constraints), but no corresponding hazard is defined in the hazard list. This means that some potential hazard and a suitable RRM for it have yet to be identified.

$$\text{Som}(\exists i(sts_i = \text{ps} \wedge (\forall x(\neg hzd_x) \vee \exists x(hzd_x \wedge \neg RRM_x))))$$

A hazard may go unnoticed for several reasons (*e.g.*, the complexity of the application which may exhibit more possible interactions than expected, or overlooked human errors). The hazards module and RRM module are iteratively updated with new recognized hazards and their appropriate RRMs.

The third issue is addressed simply by comparing the verification output before and after the introduction of the RRMs, to see if the residual risk value is negligible or not.

If and when the RRMs introduced are enough, the risk value remains negligible throughout the execution of the application and the iterative process ends.



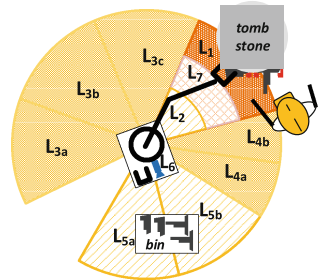
## 4 SAFER-HRC for a Case-Study

In this section the application of SAFER-HRC on a real use case is illustrated. The chosen scenario is a part of the application for preparing a machining pallet. Usually done by skilled operators, the task, depicted in Fig. 5, includes the assembly of workpieces to be machined into fixtures attached to a so-called tombstone. A small collaborative robot carrying a screwdriver end-effector is installed on a cart close to both the pallet and the operator for assisting in fixing the fixtures. The work-cell is equipped with cameras and sensors that detect the position of robot and operator. In the task, the operator is supposed to set a workpiece (wp) in place from a bin into the tombstone fixtures, and command the robot to move in from its homing position and fix the wp. The operator is supposed to hold the wp until it is fully attached to the tombstone and screwdriving is complete, in order to inspect and assess the task execution.

In this example the layout is divided into eleven regions, defined by the safety analyzer, according to the expected motion directions of robot links and the presence of potential/actual obstacles. Region  $L_1$  is where more risky actions such as screwdriving are executing, and the operator is in very close proximity to a sharp end-effector and constraining objects; thus, there are chances of entrapment and stronger safety constraints are needed. Region  $L_6$  is the robot homing position, where after screwing each wp, the robot goes back and remains idle until it receives some activation signal from the operator. The user-provided description of the task is broken into 15 different types of actions, as shown in Fig. 4. Action types 8–13 can be repeated in a loop, one iteration for each jig. Thus, if there is only one jig to prepare, the task model has 15 actions, whereas if there are two the task has 21 actions, as there are two instances of action types 8–13.

$pmr = op$	$pmr = ro$
1. move to the bin	7. move to tombstone
2. grasp a wp	9+i. EE approaches the pallet
3. bring the wp to the pallet	10+i. screw the held wp
4. put the wp on the pallet	11+i. EE retracts the pallet
5. hold wp until $sts_{13} = dn$	12+i. check the number of screwdriven jigs
6. send activation signal	13+i. if not all of the jigs are screwdriven, go back to $a_8$
8+i. prepare the jigs	
14. release wp after $sts_{13} = dn$	
15. send a resuming command	

**Fig. 4.** List of actions in the case study. Action types identified by  $\{\sim +i\}$ , belong in a loop.  $i$  is the loop iteration and cannot exceed the number of jigs.



**Fig. 5.** Layout of the case study.

**Iterative Risk Assessment.** All the reported experiments have been carried out on a 2,6 GHz Intel® core™ i5 machine. The maximum length of analyzed traces was set to 100 time instants, which is enough to complete all actions; no experiment took more than 153 seconds. The formal model and experiments can be found at [github.com/Askarpour](https://github.com/Askarpour).

The verification helps analyzers and designers to figure out safety flaws of the design before the deployment. For example the necessity of requirements “robot must go back to homing ( $L_6$ ) after screwdriving each single wp”, and “robot must remain in homing until it receives a new signal from the operator” was highlighted after multiple iterations of the verification. Thus, when operator is moving around in the layout (*e.g.*, to pick a wp from the bin), the robot is idle in the home position and no harm is threatening her.

Another contribution of the verification is to discover the errors made by the operator, which are neglected in the initial design and may raise dangerous situations. For instance, in this scenario we initially forgot to consider the following possible errors and misuses:

1. the operator mistakenly sends the activation signal to the robot before settling the part on the fixtures;
2. the operator bends down and brings her head close to the tomb while wp is being screwdriven (*e.g.*, to better monitor the procedure), just as screwdriving is about to finish and EE is about to move backwards from the tomb;
3. the operator stays on the right side of the tomb while holding the wp to be screwdriven, which can lead to the operator getting entangled between the tombstone and a robot link or to getting hit by a sweeping robot arm.

Each of these cases was highlighted by the violation of a safety constraint at some iteration. For instance, the first item was discovered when the model was unable to satisfy a safety constraint of  $a_3$  “operator brings the wp to the tomb”. This constraint requires the robot to be idle in homing,<sup>1</sup> but if the operator mistakenly sends the activation signal before she goes to the tomb, then operator and robot are both moving in  $L_1 \vee L_2 \vee L_4$ , which might lead to the operator being hit by the robot forearm link. As an RRM to avoid this,  $preC_3$  is updated with formula  $p_{hands} = L_1 \wedge signal$ , meaning that  $a_3$  starts to execute when operator is in  $L_1$  (her hands are on tomb to hold the wp) and she sends the activation signal.

After further verification iterations, the second case was highlighted by the violation of constraint  $sftC_{11}$ , which was that only the hands of operator are allowed to be close to the tombstone, but not other body parts. This can happen due to operator’s lack of awareness, experience or familiarity with the instructions. To address this situation, a new RRM (*i.e.*, a set of constraints) is added stating that if  $sftC_{11}$  is violated during the execution of action  $a_{11}$ , the latter

---

<sup>1</sup> A safety constraint is achieved by a safety function in charge of reliably accomplishing the risk reduction objective. The reliability level is defined according to analysis and methods of functional safety, as for ISO 13849, for instance. In this case the safety function is to monitor the position of motors so as to prevent unwanted motion from the desired resting position.

**Table 2.** Severity value according to the involving robot and human parts.

Hazard type	Robot part	Body part		
		Head & shoulders	Waist	Hands, arms and fingers
Hit	EE	4	2	3
	R <sub>1</sub>	3	1	2
	R <sub>2</sub>	3	1	2
Entrapment	R <sub>1</sub>	2	1	1
	R <sub>2</sub>	2	0	1

must pause, EE must go back to its starting position on the tomb surface, and the robot must stay idle until the operator sends a resumption signal.

A third case was also detected during the verification, due to similar reasons as the previous case. If operator’s position somehow blocks the way of KUKA’s links, the robot should stop its motion and remain idle until receiving a resumption signal. Thus, an RRM for emergency stop is defined, which in this case has a structure similar to an action: a pre-condition (violation of  $sftC_7$ ), a safety constraint (remaining idle), and a post-condition (safety violation is removed and resumption signal is received).

In the rest of this section, we illustrate the formalization of a fragment of the case study, focusing on two types of hazards: (i) a transient impact (*i.e.*, contact with possibility of recoiling) between the operator and KUKA parts (R<sub>1</sub>, R<sub>2</sub> and EE); and (ii) operator getting entrapped between one of KUKA’s links and some physical obstacle such as tomb or sidewalls of the wp bin. Any hazard of the two types has different severity depending on the impacted body parts (head and shoulder area, arms and hands area, waist area), the parts of KUKA and the layout shapes that are involved. Table 2 shows how severity is assigned to any detected hazard; the table groups the parts of operator’s body into three areas to ease the presentation; in addition, for simplicity we assume that the severity level of a hazard is the same in all regions of the layout.

The following formula represents an example of hit hazard and defines that hazard “head hit by R<sub>1</sub>” ( $h_1$ ) occurs when the human head area is close to at least one robot part within  $L_{3_a}$ ,  $L_{3_b}$ , or  $L_{3_c}$ .

$$hzd_{h_1} \Leftrightarrow p_{R_1} = (p_{head} | p_{neck} | p_{shoulders}) \wedge p_{R_1} = (L_{3_a} | L_{3_b} | L_{3_c})$$

For this kind of hazards, an RRM is defined such that the involving KUKA part moves away from the operator (while the robot is idle, *i.e.*, it is not executing any action) until the hazard is removed. This is formalized by the following formula, which provides the definition of RRM for  $h_1$  hazards:

$$hzd_{h_1} \Rightarrow \text{Until}_w (\text{adj}(\text{futr}(p_{R_1}, 1), p_{R_1}) \wedge \neg \exists i (sts_i = \text{exe} \wedge pmr_i = ro), \neg hzd_{h_1})$$

Similar formulae exist for combinations of R<sub>2</sub>, EE, and other body areas.

An example of entrapment hazard is “head entrapped by R1”,  $e_1$ , which can happen when the human head area is close to a KUKA part, in a section of the

layout where there are physical obstacles, and there is little room for movement due to the presence of the wp bin and of the tombstone—*i.e.*,  $L_1$ ,  $L_{5_a}$  and  $L_{5_b}$ .

$$hzd_{e_1} \Leftrightarrow p_{R_1} = (p_{head}|p_{neck}|p_{shoulders}) \wedge p_{R_1} = (L_{5_a}|L_{5_b}|L_1)$$

This hazard is treated by the RRM formalized by the following formula, which makes the robot part stop any movement when close proximity is detected.

$$hzd_{e_1} \Rightarrow \text{Until}_w (\text{futr}(p_{R_1}, 1) = p_{R_1} \wedge \neg \exists i (sts_i = \text{exe} \wedge pmr_i = ro), \neg hzd_{e_1})$$

Similar formulae are defined for hazard and RRM concerning  $R_2$ . Given the defined severity value, the risk estimator module associates a risk value with each detected hazard. If the outcome of the risk estimation analysis is that an RRM is *required* due to the presence of the hazard, then an appropriate RRM should be chosen by the safety analyzer, from among the different possibilities in case more than one RRM is available. If the outcome is that an RRM is only *recommended*, she can decide whether to introduce one or not.

## 5 Conclusion

In this paper we enriched SAFER-HRC, which is a methodology for risk assessment of collaborative robotic applications. It relies on automated formal verification techniques for uncovering potential hazards in applications under design. It is a semi-automated solution because it automatically identifies hazards and estimates their risk, yet it relies on a human analyzer to provide input information about the application and the physical layout, and to modify or update the list of hazards. In addition, it is the analyzer who decides whether to introduce or not RRMs to counter non-negligible risks, or which RRM(s) to introduce when more than one is available.

Currently we have prepared a prototype tool that transforms UML diagrams to the formal model, which relieves the safety expert willing to use SAFER-HRC from touching the formal model and formulae. We have also conducted experiments on a few more case-studies, for example a different assembly task by a KUKA arm presented in Euroc project ([www.euroc-project.eu](http://www.euroc-project.eu)).

Future work will focus on introducing probabilities in the  $O$  module which replicates erroneous human behavior and frequent mistake phenotypes.

## References

1. Zot: a bounded satisfiability checker. [github.com/fm-polimi/zot](https://github.com/fm-polimi/zot)
2. Askarpour, M., Mandrioli, D., Rossi, M., Vicentini, F.: SAFER-HRC: safety analysis through formal vERification in human-robot collaboration. In: Skavhaug, A., Guiochet, J., Bitsch, F. (eds.) SAFECOMP 2016. LNCS, vol. 9922, pp. 283–295. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-45477-1\\_22](https://doi.org/10.1007/978-3-319-45477-1_22)
3. Baresi, L., Pourhashem Kallehbasti, M.M., Rossi, M.: Efficient scalable verification of LTL specifications. In: Proceedings of Software Engineering (2015)

4. Bouti, A., Kadi, D.A.: A state-of-the-art review of FMEA/FMECA. *Int. J. Reliab. Qual. Saf. Eng.* **1**, 515 (1994)
5. Brederke, J., Lankenau, A.: Safety-relevant mode confusions modelling and reducing them. *Reliab. Eng. Syst. Saf.* **88**(3), 229–245 (2005)
6. Butterworth, R., Blandford, A., Duke, D.J.: Demonstrating the cognitive plausibility of interactive system specifications. *Formal Asp. Comput.* **12**, 237–259 (2000)
7. Dhillon, B.S., Fashandi, A.R.M.: Safety and reliability assessment techniques in robotics. *Robotica* **15**, 701–708 (1997)
8. Dixon, C., Webster, M., Saunders, J., Fisher, M., Dautenhahn, K.: “The Fridge Door is Open”—temporal verification of a robotic assistant’s behaviours. In: Mistry, M., Leonardis, A., Witkowski, M., Melhuish, C. (eds.) *TAROS 2014. LNCS (LNAI)*, vol. 8717, pp. 97–108. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-10401-0\\_9](https://doi.org/10.1007/978-3-319-10401-0_9)
9. Fu, J., Topcu, U.: Synthesis of shared autonomy policies with temporal logic specifications. *IEEE Trans. Autom. Sci. Eng.* **13**(1), 7–17 (2016)
10. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: *Modeling Time in Computing. Monographs in Theoretical Computer Science. An EATCS Series.* Springer, Heidelberg (2012)
11. Guiochet, J.: Hazard analysis of human-robot interactions with HAZOP-UML. *Saf. Sci.* 225–237 (2016). [abs/1602.03139](https://doi.org/10.1016/j.ssci.2016.03.039)
12. Guiochet, J., Do Hoang, Q.A., Kaaniche, M., Powell, D.: Model-based safety analysis of human-robot interactions: the MIRAS walking assistance robot. In: *Proceedings of ICORR* (2013)
13. International Electrotechnical Commission: IEC 61882, Hazard and operability studies (HAZOP studies)-Application guide (2001)
14. International Standard Organisation: ISO10218-2:2011, Robots and robotic devices - Safety requirements for industrial robots - Part 2: Robot Systems and Integration
15. International Standard Organisation: ISO14121-2:2007, Safety of machinery - Risk assessment - Part 2
16. International Standard Organisation: ISO15066:2016, Robots and robotic devices - Collaborative robots
17. Leveson, N.: *Engineering a Safer World: Systems Thinking Applied to Safety.* MIT Press, Cambridge (2011)
18. Machin, M., Dufossé, F., Blanquart, J.-P., Guiochet, J., Powell, D., Waeselynck, H.: Specifying safety monitors for autonomous systems using model-checking. In: Bon-davalli, A., Di Giandomenico, F. (eds.) *SAFECOMP 2014. LNCS*, vol. 8666, pp. 262–277. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-10506-2\\_18](https://doi.org/10.1007/978-3-319-10506-2_18)
19. Machin, M., Dufossé, F., Guiochet, J., Powell, D., Roy, M., Waeselynck, H.: Model-checking and game theory for synthesis of safety rules. In: *Proceedings of HASE* (2015)
20. Martin-Guillerez, D., Guiochet, J., Powell, D., Zanon, C.: A UML-based method for risk analysis of human-robot interactions. In: *Proceedings of SERENE.* ACM (2010)
21. Pouliezos, A., Stavrakakis, G.S.: Fast fault diagnosis for industrial processes applied to the reliable operation of robotic systems. *Int. J. Syst. Sci.* **20**, 1233–1257 (1989)
22. Salem, M., Lakatos, G., Amirabdollahian, F., Dautenhahn, K.: Would you trust a (faulty) robot?: effects of error, task type and personality on human-robot cooperation and trust. In: *Proceedings of ACM/IEEE Human-Robot Interaction, HRI* (2015)
23. Sharma, T.C., Bazovsky, I.: Reliability analysis of large system by Markov techniques. In: *Proceedings of the Symposium on Reliability and Maintainability* (1993)

24. Sierhuis, M., Clancey, W.J., Hoof, R.J.V.: Brahms: a multi-agent modelling environment for simulating work processes and practices. *Int. J. Simul. Process Model.* **3**, 134–152 (2007)
25. Stocker, R., Dennis, L., Dixon, C., Fisher, M.: Verifying brahms human-robot teamwork models. In: del Cerro, L.F., Herzig, A., Mengin, J. (eds.) *JELIA 2012. LNCS (LNAI)*, vol. 7519, pp. 385–397. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33353-8\\_30](https://doi.org/10.1007/978-3-642-33353-8_30)
26. Webster, M., Dixon, C., Fisher, M., Salem, M., Saunders, J., Koay, K., Dautenhahn, K.: Formal verification of an autonomous personal robotic assistant. In: *Formal Verification and Modeling in Human-Machine Systems* (2014)
27. Webster, M., Dixon, C., Fisher, M., Salem, M., Saunders, J., Koay, K.L., Dautenhahn, K., Saez-Pons, J.: Toward reliable autonomous robotic assistants through formal verification: a case study. *IEEE Trans. Hum. Mach. Syst.* **46**, 186–196 (2016)

# Multi-level Static Analysis for Finding Error Patterns and Defects in Source Code

Andrey Belevantsev<sup>1,2(✉)</sup> and Arutyun Avetisyan<sup>1,2,3,4</sup>

<sup>1</sup> Institute for System Programming of the Russian Academy of Sciences,  
Moscow, Russia

{abel, arut}@ispras.ru

<sup>2</sup> Moscow State University, Moscow, Russia

<sup>3</sup> Moscow Institute of Physics and Technology (National University),  
Moscow, Russia

<sup>4</sup> National Research University Higher School of Economics, Moscow, Russia

**Abstract.** This paper presents the formalism for multiple level static analysis for defect detection in source code. The first level has the program and memory model that are suitable for AST-level checks. The following levels address detection of critical errors: on the second level interprocedural partially context-sensitive analysis is performed via dataflow analysis and symbolic execution with state merging, whereas the third level adds path-sensitivity via predicate tracking for the dataflow information computed on the second. The analysis designer can freely choose the appropriate analysis level or their combination to check the desired program property. The presented methods are implemented in the Svace static analysis toolset. The first analysis levels for C/C++ and Java are implemented as extensions of corresponding production compilers (Clang and javac) and Find-Bugs tool plugins, while the second and third levels make the core of Svace analyzer together with 100+ implemented checkers for critical defects. The evaluation on extra large codebases of millions lines of code such as full-blown Android and Tizen OSES has shown the approach scalability and the acceptable false positives ratio (less than 40%).

**Keywords:** Static analysis · Symbolic execution · Defect detection

## 1 Introduction

Static analysis tools have become essential for improving program security with their application in the secure development lifecycle. While there are many flavors of static analyses, this use case has produced very concrete requirements for the analysis tools. The key requirements are fully automatic analysis, scalable to very large codebases yet maintaining a tolerable level of false positives, finding most well-known defect types and customizable by a power user.

When designing an analyzer adhering to these requirements, a number of important observations can be made. First, the classification of defect types shows that they demand various analysis levels. Many security related defects and coding rules need the abstract syntax tree (AST) analysis level possibly with control flow and minimal

intraprocedural data flow analysis. Common critical defects typically require interprocedural context-sensitive and path-sensitive analysis for good true positive ratio, but it also happens that interprocedural dataflow analysis based on function summaries is enough and path sensitivity is not needed. Second, making the abovementioned path-sensitive analysis for critical defects (that is needed for true positives) scalable to millions of LOCs usually means having an unsound analysis, that is, missing some of real errors. Finally, there are many analyzer components behind the core algorithms that are crucial for the success of the tools. Some of them include building the analysis intermediate representation for the supported languages/compilers transparently to the user, organizing parallel launch of the many analysis levels, providing results review APIs etc. These are of the more technical nature and will not be covered in this paper.

The research body known to us doesn't address the described issues well. Most of papers devoted to deep static analysis concentrate on variants of symbolic execution, abstract interpretation, or dataflow approaches [1–3] that explore well the possibilities of building a static analyzer with good true positives ratio, often for concrete defect types, but do not have the goal of being scalable for very large programs. Papers exploring lightweight static analyses contribute to the task of creating specialized query languages for writing checkers, of interactive analysis, or of creating checkers for specific code patterns. In addition to those issues, the question of implementing an analyzer that covers all needed defect types or of integrating existing light/heavyweight analyzers together is not addressed.

In this paper, we suggest a unified approach for building a collection of static analyzers operating on multiple levels. The first analysis level is designed for defects that require only AST level analysis with possible control flow and intraprocedural dataflow information. We suggest the memory model for C/C++ similar to the one in [5, Chap. 2.3] and [4] that allows for arbitrary pointer operations with field and array elements sensitivity. Other languages are handled here with separate analyzers, however, the approach of building a unified AST for the most popular languages (C/C++/Java/C#) [6, 7] is also possible with the same low-level memory model. This level is described in Sect. 2.

The main analysis levels deal with critical defects that require interprocedural analysis. We introduce context-sensitive interprocedural analysis with function summaries for the second level. The analysis happens on a lower level intermediate representation common to the supported languages inspired by [8, 9]. The second level analysis tracks possible values and points-to sets for variables, unifying the variables with the same values into equivalence classes (similar to a compiler value numbering, also in [8, 10]). The abstract interpretation of the function is performed with some assumptions breaking analysis soundness (non-aliasing of function arguments, loop unrolling for the fixed number of iterations). For interprocedural analysis, a function summary is created as an abstract state in the return instruction keeping up all recorded information about escaped variables' equivalence classes and values. The summary is then applied at the call instructions with substituting formal parameters with their actual values from the call context, giving reasonable level of context sensitivity. The second analysis level is described in Sect. 3.

The third analysis level introduces path-sensitivity with tracking program features as logic formulae over the variable equivalence classes and constants. Conjunction,



disjunction and limited negation are supported as logical operations. The values for variables are treated symbolically. The resulting formulae are saved in the function summaries for the interprocedural analysis as well. In the interesting program points, the additional predicate for the error condition is added, and the feasibility of the final formula is established via an SMT solver. The error is reported together with the part of the trace provided by the solver. The level is described in Sect. 4.

In Sect. 5 we briefly describe the implementation of the proposed mechanisms and experimental results. The multi-level static analysis is implemented in the Svace collection of analyzers. The first-level analysis is implemented on top of Clang Static Analyzer for C/C++ and on top of FindBugs and OpenJDK JavaC compiler for Java. A number of popular coding rules and simpler security related rules are implemented. The other levels are implemented jointly for C/C++ and Java in the Svace analysis core engine. LLVM bitcode representation and Java class files are converted into the Svace internal representation and are analyzed according to Sects. 3 and 4 with more than 100 checkers for critical defects. We have verified the implementation against source codes of Android OS versions 5, 6, and 7, and Tizen OS versions 2.2 and 2.3. The results showed the scalability of our approach (4–5 h for the full analysis of mentioned OSes) and good enough true positives level (>60%). As a result, the Svace tool collection has been deployed for use in Samsung Electronics since 2015, and the feedback provided by the company engineers has confirmed our own findings.

Section 6 concludes the paper and outlines our future plans.

## 2 Lightweight Analysis Level

The first analysis level is intended for the coding rules checking (e.g. MISRA or CERT Secure Coding standards [11, 12]), code exploiting undefined or implementation-defined behavior (e.g. potential integer overflow, division by zero, code that depends on function arguments evaluation order), various API usage checks (including security related [13], like avoiding multiple binds to the same port (CWE-615), using hash without a salt (CWE-759), etc.). The criterion for including an error type for checking on this level is the sufficiency of AST level and/or intraprocedural control and data flow analysis.

The program model used for analysis is the pair  $\langle S, F \rangle$ , where  $S$  is the per-module global symbol information (defined types and global variables) and  $F$  is the set of functions or class methods, where for each function  $f \in F$  we have the pair  $\langle A, G \rangle$ , where  $A$  is the function AST and  $G$  is its control flow graph with nodes coming from the statements contained in  $A$ . While the basic blocks are formed from the lower level statements, the links from these statements to the parent higher-level source constructs maintained in the AST allow also having some structural information about conditions, loops, switches, and exceptional flow.

The model is described for the C/C++ language, but similarly e.g. for the Java language the global information  $S$  will be class static variables and defined internal types, whereas  $F$  will describe class methods (after class flattening). Also, it is possible to reason about the model suitable for popular general purpose languages (like C/C++, Java, C#) with the formalism unifying possible global information variants,

types/classes declaration, and the unified AST having possible language operations and other structure elements as nodes [6, 7]), but for the simplicity reasons we will concentrate on the C language with the memory model that is low level enough to support the other mentioned languages.

We model the memory rather straightforward with hierarchical *memory locations* (MLs) that belong to *memory regions* (MRs). Memory regions come from the C/C++ memory classes, but for our purpose it is enough to differentiate between a global memory, a stack memory (for local variables, VLAs, and `alloca`s), and a heap memory (`malloc/new`). A memory location  $M$  is defined as  $\langle R, B, S, O, P \rangle$ , where  $R$  is the memory region,  $B$  is the base address for this location (either a virtual stack pointer, a global pointer, or a base pointer for a dynamically allocated memory chunk),  $S$  and  $O$  are the size and offset (starting from  $B$ ) for this location, respectively, and  $P$  is the parent memory location (an array for an element access, or a structure for a field access). We build the model for the intraprocedural analysis assuming type memory layout has happened, so the virtual stack pointers we use will be unique for every function and allow for disambiguating stack memory accesses with known offsets.

New memory locations are generated with the following rules:

- The local/global variable access lazily creates the memory location in the corresponding region with the size and offset known from memory layout. The exceptions for size are variable-length arrays, `alloca()` calls, and extern variables; the offset exceptions are e.g. most of array accesses with indexes known at runtime. Their size/offset values can be modeled with varying precision as described below.
- The malloc calls create the memory location with dynamic memory region, the unique  $B$  element, and with  $P = B$ . For the size element unknown at runtime, the above note for modeling applies.
- The array and field access lazily creates the child memory location with the  $P$  element set to the parent compound and the  $B$  element set to the one of the parent. The size and offset is always known for field accesses, for arrays the above note for modeling applies.

For discussing the evaluation rules for memory accesses we need to introduce the abstract values we deal with when simulating expressions. The most interesting is modeling integer evaluations and points-to sets. We suggest two choices that have a varying precision degree. The first option is to model integer expressions with the integer interval abstract domain [1] and to model points-to sets as the set of memory locations that a pointer can possibly point to. For the lightweight analysis, at most two loop iterations are considered before widening to top values or to loop upper bounds in case they are known to be constant. The second, more precise option is to utilize symbolic execution techniques, that is, to use symbolic integers adding proper guarding branch conditions at join points, and also to guard possible memory locations for pointers with branch conditions instead of doing merges on joins. This technique is possible when achieving certain intraprocedural path sensitivity is desired [4], however, the total number of traversed paths should be limited in order to avoid slowdowns that can be often observed when carelessly using path-sensitive CSA checkers.

We associate an abstract value with each memory location, and we assume that we have a *Join* function that can unify the abstract values of handled expressions or

points-to sets (either through appropriate integer range operations or through building a disjunction of possible variants with added guard predicates). The *Join* function is used on the join points in the control flow (i.e. at the entries of basic blocks with more than one predecessor) to unify values and points-to sets stored for memory locations on different paths.

Let us discuss evaluating address and dereference operators with the above modeling options in mind. The address operator `&expr` provides the memory location for `expr`, while the dereference operator `*expr` provides the abstract value equal to the *Join* over the values for memory locations of the points-to set available for `expr`. In other words, memory locations are used to evaluate `expr` as an l-value while their abstract values are used to evaluate `expr` as an r-value.

Consider the example code on Fig. 1. Assuming 32-bit integers and pointers, we'll have memory locations for `x`, `a`, `b`, and `p` with base pointer as a virtual stack pointer for `foo`, sizes equal to 32, and offsets equal to 0, 32, 64, and 128, respectively. Initially, the `x` parameter has an unknown value, the `p` pointer has an uninitialized points-to set. After evaluating the assignments to literals on line 2, we lazily initialize memory locations and values for `a` and `b`. With joining points-to sets after processing lines 4 and 6 (at the beginning of line 7), we have that  $PT(p) = \{ML(a), ML(b)\}$ , where  $PT$  is the points-to set and  $ML$  is the memory location. Then the dereference operator at line 7 gives us the return value as  $Join(Val(ML(a)), Val(ML(b))) = [7, 10]$  for the integer range abstraction. In case of guarded predicates abstraction we'll have  $(p) = (x > 4) ? ML(a) : ML(b)$  and correspondingly the return value as  $(x > 4) ? Val(ML(a)) : Val(ML(b)) = (x > 4) ? 7 : 10$ .

```
(1)  int foo(int x) {
(2)      int a = 7, b = 10, *p;
(3)      if (x > 4)
(4)          p = &a;
(5)      else
(6)          p = &b;
(7)      return *p;
(8)  }
```

**Fig. 1.** Memory locations example.

The overall analysis algorithm on the lightweight level is organized in two stages for each function. The first stage performs the AST traversal for the checkers that are only interested in AST properties. Checkers can register their interest in certain types of AST nodes (functions, loops, calls etc.) and then the traversal will notify the checker upon reaching the desired node, and when in the callback, the checker is free to move along AST by itself, too. The second stage is for the checkers that require control flow or data flow information. The control flow graph for the function is built, and then it is traversed to build the described memory model with creating memory locations lazily (upon the first use) and evaluating abstract values with the selected precision. When the model is constructed, the checker can traverse the control flow graph as desired and consult the computed values and memory locations.

To finalize the description, we would like to note that in general the performed intraprocedural analysis for calculating the memory model is sound. The unknown/top values for integer ranges are set and handled conservatively as well as the coarsening of the predicates. The unsoundness kicks in on the higher analysis levels.

### 3 Interprocedural Analysis Level

Further analysis levels are designed for interprocedural context-sensitive analyses and critical defects such as buffer overflows, null pointer dereferences, and resource leaks. Unlike the previous level, the analysis is presented with the set of functions and global information (variables, classes). The top-level analysis algorithm looks like follows:

1. Build the global program call graph. The function bodies are scanned in order to pick up the call instructions. Function pointer calls are initially considered as calls to unknown functions. Optionally, the very quick analysis is performed in order to support simple cases of single-target function pointer calls. During the function body scan, only the expressions taking function address, pointer assignments, and function pointer calls are considered. Likewise, the global variables of function pointer types are considered. In case of the function pointer call, if the pointer always equals to the single function address, the call is resolved to this address and the call graph is updated;
2. Break any cycles found in the call graph by removing an arbitrary edge that are in the cycle. Our experience shows that the precision loss arising from this decision is minimal. The alternative solution is to analyze such cycles twice in order to take into account the recursion effects;
3. Perform the intraprocedural analysis from bottom to top (in reverse topological order) according to the approach outlined in Sect. 3.1. Every function is analyzed once and the analysis effect is captured in the function *summary* (resume). The rules for creating callee summaries and applying them within the caller are outlined in Sect. 3.2.

The analysis operates on the lower-level 3-address intermediate representation suitable for the general purpose languages (C/C++/Java) similar to the LLVM IR with the attached information about high-level language types and other constructs as needed (for example, functions pointer calls in the IR that have originated from virtual calls are marked as such with on the side information). Local variables that do not have their address taken are lowered to *pseudoregisters* – temporaries that never alias with each other and are also used for evaluating complex expressions.

#### 3.1 Intraprocedural Analysis

The goal of intraprocedural analysis is to track abstract values and to compute the memory model for the single function. The memory is modeled with the memory location abstraction outlined in Sect. 2 for the lightweight analysis, that is, the analysis computes the mapping  $Mem : Vars \rightarrow MLs$  for each program point, where *Vars* are program variables and *MLs* are memory locations. Pseudoregisters get their memory locations with the special register memory region and just their size initialized.

The values that are stored in memory locations are tracked with the *concept of value classes* (VCs). It has been noticed (by us and the other researchers as well [8, 10]) that tracking value equivalences between variables plays the important role in achieving the good analysis quality. Thus, we associate a value class with each memory location (calculating the function  $Val : MLs \rightarrow Vals$ ) that acts as an abstract value and is assigned using the rules similar to assigning value numbers in the hash-based value numbering techniques [14]. The attributes of interest are then calculated as being the property of value classes, not memory locations.

Consider the code example on Fig. 2. Let the memory location for  $x$  have the value class  $vc_1$  (i.e.  $Val(ML(x)) = vc_1$ ), then we can determine that  $Val(ML(a)) = vc_2$  on line 2 and  $Val(ML(b)) = Val(ML(c)) = vc_3$  on lines 3 and 7, respectively. As the points-to set of  $Val(ML(p))$  on line 10 equals to  $\{ML(b), ML(c)\}$ , we can determine that the return location always has the value class  $vc_3$ . In case of attaching the value tracking attributes to memory locations (e.g. integer ranges), we can only find that both  $b$  and  $c$ , and correspondingly the return value will have the same integer range, but not that they will be exactly equal.

```

(1)   int bar (int x) {
(2)       int a = x + 2, *p;
(3)       int b = a + 1;
(4)       if (x > 0)
(5)           p = &b;
(6)       else {
(7)           int c = x + 3;
(8)           p = &c;
(9)       }
(10)      return *p;
(11)  }
```

**Fig. 2.** Value classes example.

The VC propagation happens as follows. For assignment operations, the VC for the left side memory location is set to the VC of the right side. For usual arithmetic and logical operations, VCs of operands are retrieved and the operand order for communicative operations is canonicalized so that the operand with the lower numbered VC goes first. Then, if the VC for this operation with these given VCs as operands has been already assigned, the same VC is returned, otherwise the new VC is created. At control flow join points the VC is assigned via  $JoinVC(vc_1, vc_2)$  function: if the memory location has the same VC along both incoming edges, then this VC is retained, otherwise the new VC is returned.

The analysis allows to calculate interesting attributes attaching them to value classes<sup>1</sup>. When simulating an instruction, the analysis updates the resulting VC and then proceeds with updating all attributes attached to the VCs of operands. The attributes that are always calculated are integer value tracking and points-to sets. Checkers typically work by defining their specific attributes and their propagation rules so that the generic VC propagation algorithm will also update these attributes.

The default integer value tracking is with the integer range abstraction as discussed in Sect. 2. The points-to tracking also follows the intraprocedural analysis described there: the address operator results in returning the memory location of its operand, control flow joins result in joining the points-to set attached to VCs of the memory location that come from different execution flows. The dereference operator retrieves the points-to set attached to the VC of its memory location. When doing a load, if the points-to set has only one element, its VC is returned, otherwise the VCs are joined pairwise with *JoinVC* together with their specific attributes (when the checker supplies the join function). When doing a store, having a single element in the points-to set allows making a strong update, otherwise for each ML from the points-to set we update its VC as *JoinVC*( $vc_{old}, vc_{new}$ ).

Loops are handled with prior natural loop detection and then simulating the loop for a fixed amount of iterations (typically three times). After all iterations have been simulated, the resulting VCs are obtained as a join over the values for all iterations, and their attributes would typically take into account loop exit conditions as well as the results of induction variable analysis. Checkers may supply specific propagation functions for the loop exit edges in order to be able to apply the widening operators [1] to their attributes as they deem necessary.

The interprocedural analysis is designed as unsound for scalability, and the main unsoundness sources are the abovementioned loop handling with analyzing limited amount of iterations and the assumption of non-aliased function arguments that works well for real life projects.

### 3.2 Creating and Applying Function Summaries

The interprocedural analysis in scalable static analyzers is commonly built around the *function summary* concept, the idea of capturing the necessary information for the analyzed function and then reusing it when processing calls to the function without having a need to touch its body again. We also utilize this idea. We build the function summary as the part of our *Mem* and *Val* mappings (with the attributes attached to VCs) computed as joins over the function return points. The part interesting to us is the one describing the function changes to its “external” memory (global variables, arguments, return values) and other memory locations that escape via the external memory links. The maximum link depth that the location should have in order to be

---

<sup>1</sup> Checkers that are more concerned with memory properties (like memory leaks) get benefit from attaching attributes directly to memory locations instead of value classes [8], but we do not discuss it in more details in this paper.

added to the summary may be limited, and this brings another source of the analysis unsoundness on the interprocedural level. The summary is built using the following algorithm:

1. Add MLs for function parameters, global variables, and return values to the summary; set  $k = 0$ .
2. For all MLs from the summary, do:
  - a. Retain the VCs of the MLs in the summary together with its attributes.
  - b. If the ML corresponds to the pointer type and has the points-to set attached to its VC, add all MLs from this points-to set to the summary.
3. Set  $k = k + 1$ . If  $k$  has reached the maximum depth level, stop, else go to step 2.

When applying the function summary to the caller’s context, we need to build the correspondence between formal and actual parameters for the function in order to be able to replace the memory locations that are stored in the summary with the locations of the caller function. We start with replacing MLs of the formal parameters that are stored in the summary with the MLs of the actual parameters, and then we update any summary MLs that have the replaced MLs as parents (their  $P$  fields). We can meet conflicts in the initial replacement when the actual parameters happen to have the same MLs (i.e. they alias each other), in which case we need to unify their VCs with *JoinVC* as well as the attached attributes. Except of this case, the value classes that are stored in the resume can be safely transferred to the caller context.

However, for other attributes attached to VCs it is desirable to redo all computations that resulted in the final attribute value based on the attributes of the actual parameters in order to gain the context sensitivity. The reason for that is when starting the simulation with unknown formal parameter values in the callee, we are likely to lose information when joining any other nontrivial attribute value with the unknown one. In order to perform such repeated computation, it is enough to remember the history of VC changes (from joins or other simulations) that are originate from function parameters’ VCs. This is because attributes clarify some properties of the value class and cannot change without changing the underlying VC (as by definition the same VCs have the same runtime value and thus cannot have different properties).

We call the initial unknown VCs of the function parameters and global variables *initial VCs*. When doing any VC update that has an initial VC as one of the parameters, we remember the source VCs and the exact operation that was performed. This information is stored alongside the resulting VC with its attributes and is therefore the part of the summary. When the MLs of the summary are replaced with the actual caller MLs, we take their initial VCs, set their (unknown) attributes as equal to the attributes of the caller’s VC, and start redoing the computation on their attributes along the saved history of changes to receive the updated attribute values for all VCs.

Consider the code presented on Fig. 3. When analyzing function *baz*, we create MLs for  $p$  and  $*p$ , the initial VC for  $ML(*p)$  as  $vc_1$ , and with corresponding different VCs for  $ML(*p)$  at lines 3 and 5 ( $vc_2$  and  $vc_3$ , respectively), reflecting that the value has been changed. However, when using the integer range abstraction, the simulation of additions on those lines and the following merge at the join point at line 6 ( $vc_4$ ) does

```

(1)  int glob;
(2)  void baz (int *p) {
(3)      *p += 2;
(4)      if (glob > 3)
(5)          (*p)++;
(6)  }
(1)  void faz () {
(2)      int k = 7;
(3)      baz (&k);
(4)      ...

```

**Fig. 3.** Applying summaries.

not add anything to the initial unknown integer range. Later, when applying the summary for *baz* at line 3 of function *faz*, we replace the ML of *\*p* from the summary with the ML of *k* from *faz*. Then, we proceed to find out that the range for initial  $vc_1$  is  $[7, 7]$ , for  $vc_2$  it is  $[9, 9]$ , and for  $vc_3$  and  $vc_4$  it is  $[10, 10]$  and  $[9, 10]$ , respectively<sup>2</sup>. This final VC value is set as the VC for  $ML(k)$  as the result of the summary application.

## 4 Path-Sensitive Analysis Level

The interprocedural level described in Sect. 3 already works well enough for a variety of checkers. Some notable examples include, first, null pointer dereference checkers that catch situations when the pointer has been explicitly compared with or assigned NULL value, and it is dereferenced unconditionally or under the same condition as the comparison. The latter constitutes the immediate runtime error, while the former can indicate either the forgotten NULL check or the excessive defensive checking. Second, tracking tainted values and sanitization code surprisingly does not always require path sensitivity, as the forgotten sanitization usually happens on all paths. Finally, specific checkers like correct usage of proper `new[]/delete[]` operators or leaving uninitialized fields in constructors require interprocedural analysis but not path sensitivity.

However, for most of buffer overflows, complex null pointer dereferences, use after free access, resource leaks, and uninitialized variables having path sensitive data is crucial. The framework described in Sect. 3 can be adapted to serve for this with the even higher precision analysis level.

Let us consider how the analysis components should be changed to achieve path sensitivity. The memory model of memory locations stays intact. The value classes become the symbolic variables that are assigned the same way through usual operations, but these operations are also stored to track the integer values of VCs. For the case of joining the VCs from different flow edges, the *JoinVC* function, when generating the new VC in case its input VCs turn to be unequal, stores the conditions on

<sup>2</sup> In case we have nontrivial VC for `glob` variable, we can further improve the precision if e.g. the conditional operator will never be executed and we don't have to consider the instruction at line 5.



which the new VC equals to its first or second input VC. The conditions are expressed as simple comparisons with constants or similar boolean operations. The points-to sets are also merged with recording conditions on which the memory location with this VC points to one or the other location. When processing the dereference operator, the points-to sets conditions are tested on satisfiability with the help of SMT solver in order to remove the unfeasible variants.

Checkers can attach the attributes to VCs that are expressed in terms of logical formulas over VCs or constants. Independently, the analysis tracks the *path predicate* that shows the condition necessary to reach the current instruction starting from the function entry. When joining the attributes from different control flow edges  $e1$  and  $e2$ , their formulas are augmented with the path predicate for the corresponding edges, so that the resulting formula is  $Path(e1) \wedge Form(e1) \vee Path(e2) \wedge Form(e2)$ , where *Path* is the path predicate and *Form* is the formula for the edge. When the checker is interested in verifying the attribute value and issuing a warning, it builds the formula out of the attribute formulas, the path predicate, and the error condition and feeds it to the SMT solver. In the case the formula is satisfiable, the warning is issued. The checker can also build an execution trace through the function to explain the warning by conjuncting the error condition formula with the control flow split conditions and giving hints at the possible execution flow.

Creating and applying function summaries for the interprocedural analysis is similar to Sect. 3.2 with respect to memory locations and value classes. The computations that are redone with initial VCs substitute the VCs for actual parameters into the attribute formulas and their conditions as well. The resulting formulas that are passed to the caller context are simplified in order to reduce their size.

Consider again the code on Fig. 3. When analyzing function *baz*, we track that  $vc_2$ 's value is actually  $vc_1 + 2$ ,  $vc_3$ 's value is  $vc_2 + 1$ , and  $vc_4$  is  $glob > 4 ? vc_2 + 1 : vc_1 + 2$ . When applying the summary for *baz* at line 3 of function *faz*, we update the final VC value as  $glob > 4 ? 10 : 9$ .

## 5 Svace Analyzer Collection

Over the course of years we have implemented the above multilevel static analysis model within the Svace analyzer, effectively turning it into the collection of analyzers, serving all three of described analysis levels for C, C++, Java [15–17], and C# [18]. The overall architecture of our implementation is depicted on Fig. 4. The first analysis phase is source code parsing and the intermediate representation construction for the further phases. This process should be transparent to the user, so it is implemented as an OS-dependent build monitoring component capturing the compilation, assembly, linkage, and other events of the original build process and then passing it to the Python library. The library organizes running our own compilers for each supported languages and/or running the 1<sup>st</sup> level static analyzers. For C/C++, the compiler/analyzer is based on the heavily patched Clang/CSA [20] 3.8 (more than 2000 patches for C/C++ dialects

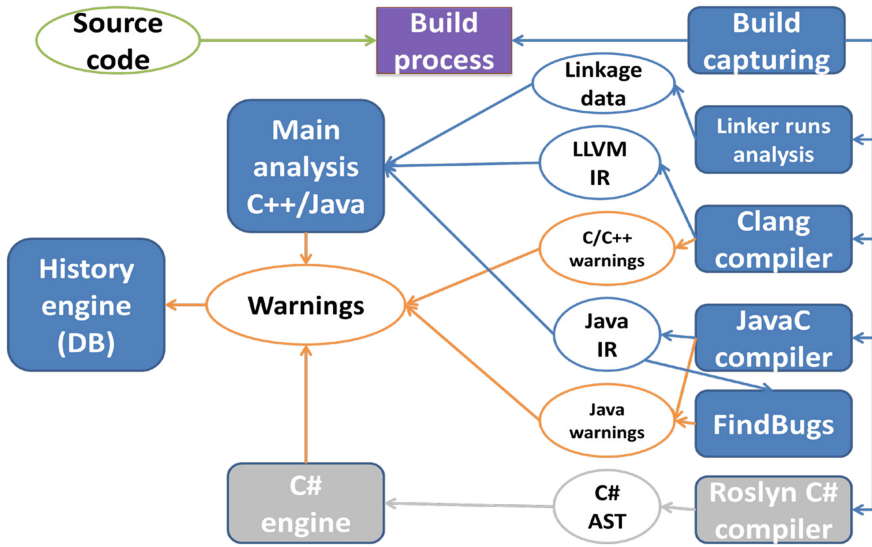


Fig. 4. Svace analyzer collection architecture.

compatibility and the lightweight checkers<sup>3</sup>). For Java, we use the OpenJDK 8-based `javac` compiler with extra patches for better compatibility with earlier Java versions and for syntactic-based checkers. Most of the other 1<sup>st</sup> level static checkers are implemented in the FindBugs tool infrastructure [19]. Also, we additionally capture and save all JAR libraries used in the original compilation. The C# checkers at all levels are based on Roslyn and implemented separately from the main engine [18].

The second, main analyzer phase is parsing available language-dependent IR (LLVM IR or Java bytecode classes) into the Svace IR and then proceeding, separately for each language, with the top-level analysis algorithm sketched in Sect. 3. It is implemented in Java. For C/C++ analysis, we additionally capture linker invocations in order to be able to resolve external function calls reliably; when this information is not available, own heuristics are used (includes information, source file paths). The interprocedural analysis starts with analyzing functions that have *specifications* – short codes capturing the essence of what the function does and is interesting for the analysis. Specifications are needed to provide the knowledge about the standard libraries placed in the program environment, which we don’t expect to have the source code for. They are written on the program’s language (C/C++ or Java) using special Svace API calls (e.g. to tell the analyzer that this function argument is a buffer and that one is its length) and compiled to the Svace IR just like any other function. For Java, after specifications we also analyze the captured JAR libraries.

The analysis core is responsible for calculating the memory model (memory locations and their value classes) on two analysis levels simultaneously (with integer

<sup>3</sup> We do not use the CSA path-sensitive infrastructure, as its scope is limited by the compilation unit, but move the heavy checkers to our own analyzer engine.

range/points-to sets tracking and merging data or additionally remembering predicates at the join points). Checkers are responsible on creating their own attributes and attaching them to value classes. The APIs are provided for querying the formula satisfiability via Z3 solver [21] and for issuing warnings. We have implemented more than 150 checkers for more than 20 critical defect types.

The intraprocedural analyses are run in parallel with the configurable number of threads. Any functions that don't have a path in the call graph connecting them can be analyzed in parallel. The thread pool manager tries to interleave reading the function bodies with the analysis and to avoid storing lots of function summaries in memory – once all function callers have been analyzed, its summary is no longer needed. The analyzer can serialize summaries to disk upon hitting the user-configurable memory consumption limit or can turn off the serialization completely to speed up the analysis in the conditions of large memory space available.

Upon the analysis completion warnings from all analysis levels are stored in the database in the unified format (warning type, message, locations, source/sink and extended explanatory execution trace, if available). The database also stores the tokenized program source code in order to be able to present the analysis results to the user in the web-based GUI. The user review results (whether the warning is a false positive, a true positive, or intentionally written, together with the user comments) are also stored. All this data constitutes the single *analysis run*, and the GUI provides the ability of comparing different analysis runs so that the user can check only newly appeared warnings also not seeing again the warnings that were once classified as false positives. This functionality is crucial for the production deployment of the analyzer. The other important supported use case is so-called *fast analysis* – the possibility to reanalyze the changed source files while using the function summaries from the previous full analysis run for the unchanged code<sup>4</sup>, but its design and implementation are out of scope of this paper.

We have evaluated the Svace analyzer collection on a variety of open source projects, ranging from thousands to millions of LOCs. The analysis time for large projects we tried is under 5 h for a machine with 32 threads and up to 90 Gb memory occupied [8] (Android 5 with 8.4MLOC – 4 h 55 min, Tizen 2.3 with 6.5 MLOC – under 3 h, Linux kernel version 3.17 – about an hour; the Java analysis usually is faster and for Android 5 Java takes little more than an hour). The analysis quality is evaluated with own test suite of more than 2000 self-contained tests and by constantly reviewing the analysis results for the large projects and providing markup for the warnings found. For Android 5, our evaluation shows 70-80% true positives for null dereference warnings, ~60% true positives for buffer overflow warnings [23], ~50% for memory leaks and ~60% for resource leaks [8].

Since 2015, Svace has been deployed in Samsung Electronics development process and for now analyzes most of company's source code [22]. The analysis quality results gathered by us have been independently confirmed by the company's engineers.

---

<sup>4</sup> One example is the Android app fast analysis that takes into account the calculated analysis data for the full Android OS.

## 6 Conclusions

We have presented a design for the multilevel static analysis, including a memory model, a value model for basic intraprocedural, an interprocedural analysis, and a path-sensitive analysis. The multilevel static analysis allows implementing the checkers for all of the defects required in production by utilizing the appropriate analysis levels, or even combining the results of simple yet high true positive rate checker capturing “easy” widespread error patterns with the more complex checker dealing with path-sensitive situations. The lower level IR used for the core analysis level and the flexible memory model allows supporting different general purpose languages in the analyzer. Putting the most important data about pointers and integer values into the analysis core also allows for scalability and extensibility (adding new checkers does not influence significantly the total analysis time).

Except the usual process of constantly refining the analysis engine and checkers implementation, we are directed now towards the better reflection of the high-level properties of object-oriented languages in the analyzer, improving the fast analysis capabilities in order to be able to easily integrate the analysis into continuous integration systems, and improving the data structures representing the logic formulas and their simplification approaches for the path-sensitive analysis level<sup>5</sup>. We are also experimenting with completely orthogonal approaches such as utilizing machine learning techniques for separating true positive warnings out of false positive ones in the analysis results presentation for the user.

## References

1. Cousot, P., Cousot, R., Feret, J., et al.: Why does Astrée scale up? *Form Methods Syst. Des.* **35**, 229 (2009). <https://doi.org/10.1007/s10703-009-0089-6>
2. Xie, Y., Aiken, A.: Saturn: a scalable framework for error detection using Boolean satisfiability. *ACM Trans. Program. Lang. Syst.* **29**(3), 1–43 (2007). Article 16
3. Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program analysis using symbolic ranges. In: Nielson, H.R., Filé, G. (eds.) *SAS 2007*. LNCS, vol. 4634, pp. 366–383. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-74061-2\\_23](https://doi.org/10.1007/978-3-540-74061-2_23)
4. Xu, Z., Kremenek, T., Zhang, J.: A memory model for static analysis of C programs. In: Margaria, T., Steffen, B. (eds.) *ISoLA 2010, Part I*. LNCS, vol. 6415, pp. 535–548. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-16558-0\\_44](https://doi.org/10.1007/978-3-642-16558-0_44)
5. Ignatiev, V.: Using static analysis for customizable checks of C/C++ semantic constraints. Ph.D. thesis, Moscow (2015)
6. Strein, D., Kratz, H., Lowe, W.: Cross-language program analysis and refactoring. In: *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, Washington, DC, USA, pp. 207–216. IEEE Computer Society (2006)
7. Zubov, M.V., Pustygin, A.N., Startsev, E.V.: Use of the intermediate software representations for static analysis of source code. *Doklady TUSUR* **27**, 64–68 (2013). (in Russian)

---

<sup>5</sup> The predicate representation has to be regularly reviewed as the path-sensitive analysis portion in the analyzer grows over time.

8. Borodin, A.: Interprocedural context-sensitive static analysis for finding defects in C/C++ program source code. Ph.D. thesis, Moscow (2016)
9. LLVM Language Reference Manual. <http://llvm.org/docs/LangRef.html>
10. Tucker Taft. The use of value numbers in static analysis. <http://www.adacore.com/knowledge/technical-papers/the-use-of-value-numbers-in-static-analysis/>
11. MISRA C 2012 Guidelines. <https://www.misra.org.uk/MISRAHome/MISRAC2012/tabid/196/Default.aspx>
12. CERT C Coding Standard. <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
13. CWE, Common Weakness Enumeration Database. <https://cwe.mitre.org/>
14. Briggs, P., Cooper, K.D., Taylor Simpson, L.: Value numbering. *Softw. Pract. Exper.* **27**(6), 701–724 (1997). <https://doi.org/10.1002/zaac.201500219>
15. Borodin, A., Belevantsev, A.: A static analysis tool Svace as a collection of analyzers with various complexity levels. *Trudy ISP RAN/Proc. ISP RAS* **27**(6), 111–134 (2015). (in Russian)
16. Ivannikov, V.P., Belevantsev, A.A., Borodin, A.E., Ignat'ev, V.N., Zhurikhin, D.M., Avetisjan, A.I., Leonov, M.I.: Sticheskiy analizator Svace dlja poiska defektov v iskhodnom kode programm [Svace: static analyzer for detecting of defects in program source code]. *Trudy ISP RAN [The Proceedings of ISP RAS]* **26**(1), 231–250 (2011). [https://doi.org/10.15514/ispras-2014-26\(1\)-7](https://doi.org/10.15514/ispras-2014-26(1)-7). (in Russian)
17. Avetisjan, A.I., Belevantsev, A.A., Borodin, A.E., Nesov, V.S.: Ispol'zovanie sticheseskogo analiza dlja poiska ujazvimostej i kriticheskikh oshibok v iskhodnom kode program [Using static analysis for searching vulnerabilities and critical errors in the source code of programs]. *Trudy ISP RAN [The Proceedings of ISP RAS]* **21**, 23–38 (2011). (in Russian)
18. Koshelev, V.K., Ignatyev, V.N., Borzilov, A.I.: C# static analysis framework. *Trudy ISP RAN/Proc. ISP RAS* **28**(1), 21–40 (2016). (in Russian)
19. FindBugs tool. <http://findbugs.sourceforge.net/>
20. Clang Static Analyzer. <http://clang-analyzer.llvm.org/>
21. Z3 Theorem Prover. <https://github.com/Z3Prover/z3>
22. Svace tool deployed in Samsung. *Vedomosti news*. <http://www.vedomosti.ru/technology/articles/2016/11/17/665253-russkie-programmisti-samsung>
23. Dudina, I.: Inter-procedural buffer overflows detection in C/C++ source code via static analysis. *Trudy ISP RAN/Proc. ISP RAS* **28**(5), 119–134 (2016). [https://doi.org/10.15514/ispras-2016-28\(5\)-7](https://doi.org/10.15514/ispras-2016-28(5)-7). (in Russian)

# Pipelined Bottom-Up Evaluation of Datalog Programs: The Push Method

Stefan Brass<sup>(✉)</sup> and Heike Stephan

Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg,  
Von-Seckendorff-Platz 1, 06099 Halle(Saale), Germany  
{brass,stephan}@informatik.uni-halle.de

**Abstract.** In this paper, we present a method for bottom-up evaluation of Datalog programs in deductive databases that “pushes” derived facts immediately to other rules where they are used for deriving more facts. In this way, the materialization of derived relations is avoided as far as possible. Derived facts are represented by values in variables and a location in the program, and not as explicitly constructed tuples. This helps to avoid copying operations and to keep the actively used memory small to make better use of modern processors. The method can be quite easily explained by translating Datalog to functions in a standard programming language like C++ and then using optimizations of the C++ compiler like inlining. First performance tests with benchmarks from the OpenRuleBench collection give good results. This is interesting because systems based on SLD-resolution with tabling such as XSB have beaten older deductive database implementations based on bottom-up evaluation. Now it seems that bottom-up evaluation can be done in a very competitive way.

## 1 Introduction

Database application programs usually consist of queries, written in a declarative language (typically SQL), and surrounding program code, written in a procedural language (like PHP or Java). The goal of deductive databases is to write a larger part of the application, ideally the entire program, in a declarative language based on Prolog/Datalog.

This has many aspects, e.g. the invention of new language constructs for declarative output [4]. However, the classic task of improving the speed of query evaluation is still important. The amount of data to be processed is ever growing. We might also be able to make better use of new processor architectures. For our approach, which is implemented by translating Datalog to C++, also the compiler technology is very important. The optimization in a compiler has become more powerful, so approaches that were too slow earlier can be very competitive now.

In this paper, we further develop the “Push Method” for bottom-up evaluation introduced in [3,6]. It applies the rules from body to head (right to left) as any form of bottom-up evaluation, but it immediately “pushes” a derived fact to

other rules with matching body literals. In contrast, the classic approach would first apply a rule completely before using the derived facts (which also requires intermediate storage of these tuples). Our method has the following advantages:

- We try to avoid the materialization and storage of facts as far as possible, and on a lower level the copying of values. For instance, consider the rule

$$p(X, Z) \leftarrow q(X, Y) \wedge r(Y, Z).$$

Suppose that we have derived a new fact  $q(a, b)$  and let  $r$  be a database relation (defined by facts). Now if there are many facts  $r(b, Z)$ , there is no reason to copy the value  $a$  of  $X$  for each such fact. Actually, standard implementations of SLD-resolution/Prolog would also not touch the value of  $X$  while they backtrack over different solutions for the second body literal, whereas standard implementations of bottom-up evaluation do this.

- By immediately using a derived fact, we can keep the data values near to the CPU (in registers or the cache), and thereby speed up the computation. Furthermore, memory is used for a shorter time and can be recycled earlier.
- Another feature of our method is that it does partial evaluation and rule specialization at “compile time”. The method translates a given Datalog program to C++ (which is then compiled by a standard compiler). We assume that the rules defining derived predicates are known in this step, whereas the facts for the database predicates (including user input) are known only at runtime. Time invested in the compilation phase can later be redeemed over many executions of the resulting program (with different database states). Actually, it might pay off even in a single execution, because often there are many more database facts than rules.
- Finally, if there is large number of mutually recursive rules, classic bottom-up evaluation would iteratively apply all these rules until none produces a new fact. If only a small number of these rules can actually fire in an iteration (because only they have new matches for body literals), this is obviously inefficient. Our “Push Method” does not look at rules unless there really is a new matching fact for a body literal.

The main contribution of the current paper, compared with our older papers, is that we now generate a set of recursive procedures and rely on the optimizations of the compiler (in particular, inlining and copy propagation). Instead, the version of [6] used C++ only as a portable assembler and generated a single large procedure with a big `switch`. This implementation managed its own stack, but only for recursive rules. We thought that the procedure call overhead would be too large. However, because the compiler unfolds procedure calls (“inlining”), this is not necessarily the case. We did compare the runtime on a number of benchmarks from the OpenRuleBench collection [11], and the runtime was about the same, sometimes even slightly faster. One reason for improved runtime might be that the older version produced code for each specialized version of a predicate only once, while a compiler that does aggressive inlining is stopped only by recursions—it may inline several calls to the same procedure if the body is

not large. This saves jumps and stack usage. The new version of the method is also much simpler to understand than the old one.

## 2 Query Language

We consider basic Datalog, i.e. pure Prolog without negation and function symbols. Thus, a logic program is a finite set of rules of the form  $A \leftarrow B_1 \wedge \dots \wedge B_n$  with atomic formulas  $A$  (the head literal of the rule) and  $B_1, \dots, B_n$  (the body literals of the rule). The atomic formulas have the form  $p(t_1, \dots, t_m)$  with a predicate  $p$  and argument terms  $t_i$ , which are variables or constants. As usual in Prolog,  $\leftarrow$  is written “:-”,  $\wedge$  is written “,”, and variables start with an uppercase letter to distinguish them from constants. As an example, consider

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
answer(Z) :- grandparent(sam, Z).
```

This logic program computes the grandparents of **sam**, given database relations **mother** and **father**. We assume that there is a “main” predicate **answer**, for which we want to compute the derivable instances. All other predicates are views or names for subexpressions of the query.

We require range-restriction (allowedness), i.e. all variables in the head of the rule must also appear in a body literal. This ensures that when rules are applied bottom-up (from right to left), only variable-free atomic formulas (facts) are derived. As usual in deductive databases, the predicates are classified into

- EDB-predicates (“extensional database”), which are defined only by facts (usually a large set of facts stored in a database or specially formatted files). In the above example, **mother** and **father** are EDB predicates.
- IDB-predicates (“intensional database”), which are defined by rules. In the above example, **parent**, **grandparent** and **answer** are IDB predicates.

We also say “EDB body literal” for a body literal with EDB-predicate, and “IDB body literal” for one with IDB-predicate.

Since we use a compiler-based approach, one would probably not want to compile the constant “**sam**” into the program, but to execute the compilation result many times for computing the grandparents of different people. This can easily be achieved by replacing the last rule by

```
answer(Z) :- input(X), grandparent(X, Z).
```

Here, **input** is a new EDB-predicate which is set before each execution, and contains e.g. the single fact **input(sam)**. The result of the compilation depends only on the rules (and possibly facts) for the IDB-predicates. The database state, i.e. the extensions of the EDB-predicates, can be arbitrarily updated.



### 3 Goal-Directed Query Evaluation with SLDMagic

Pure bottom-up evaluation is not goal-directed, i.e. it computes all derivable facts without looking at the given query. It is standard to do the magic sets program transformation [1] first, which reduces the applicability of rules such that only facts relevant to the given query can be derived. The output of this transformation is again a Datalog program, which is then used as input for bottom-up evaluation.

In our case, we use an improved “Magic Set” method, SLDMagic [2,5]. This is interesting because it simulates SLD-resolution (the algorithm underlying Prolog) more precisely than Magic Sets (furthermore, the output is mostly linear Datalog, which is good for the Push method, see below). Basically, the rules resulting from the SLDMagic transformation compute the nodes of the SLD-tree.

*Example 1.* The result of the SLDMagic prototype<sup>1</sup> for the above program is (with slight renamings of variables):

```
p0(X) :- input(X).
p4(Y) :- p0(X), mother(X,Y).
p4(Y) :- p0(X), father(X,Y).
p7(Z) :- p4(Y), mother(Y,Z).
p7(Z) :- p4(Y), father(Y,Z).
answer(Z) :- p7(Z).
```

A fact of the form `p0(X)` corresponds to the goal `grandparent(X,Z)` in the SLD-tree, where `Z` is the result variable (i.e. the variable appearing in the original query). Since SLDMagic must produce range-restricted rules, only variables that are bound to a constant appear as arguments in the generated predicates like `p0`. A fact of the form `p4(Y)` corresponds to the goal `parent(Y,Z)` in the SLD-tree, still with `Z` as result variable. I.e. if one finds a `Z` with `parent(Y,Z)`, this `Z` is an answer to the original query.

SLD-Resolution does several steps in between, which have been removed by the “copy rule elimination” phase of the SLDMagic prototype (this explains why there are holes in the predicate numbers). Finally, `p7(Z)` corresponds to the empty goal in the SLD-tree, i.e. the query is proven, and `Z` is a result value.

### 4 The Push Method with Procedure Calls

Whereas Magic Sets and SLDMagic are translations from Datalog to Datalog, the Push method is implemented here by a translation from Datalog to C++. The resulting program is then compiled to machine code by a standard optimizing compiler. In our tests, we have used `g++`. Because the C++ compiler performs powerful optimizations, such as inlining and copy propagation, the code we generate from a Datalog program can be simple.

<sup>1</sup> Available at <http://users.informatik.uni-halle.de/~brass/sldmagic/>.

## 4.1 Basic Code Structure, Requirements

Because we want to “push” derived facts through rules (from a body literal to the head), rules with a single IDB body literal (where derived facts can match) are especially simple:

**Definition 1 (Rule Classification).** *Rules in the given Datalog program are classified into*

- *Start rules: Rules with only EDB-body literals,*
- *Simple (or “linear”) rules: Rules with exactly one IDB body literal,*
- *Complex rules: Rules with more than one IDB body literal.*

A Datalog program without complex rules is a linear Datalog program. Our SLDMagic transformation produces programs that are mostly linear (only for rules that are recursive, but not tail recursive, complex rules are generated).

The C++ program that is generated from a given set of rules basically contains for each IDB-predicate  $p$  of arity  $n$  a corresponding procedure  $\mathbf{p}$  with  $n$  arguments that is called (at least) once for each derivable fact  $p(c_1, \dots, c_n)$ . If one looks at the procedure calls ordered by the time when they occur, one gets the following notion of a computation sequence:

**Definition 2 (Computation Sequence).** *Let a Datalog program  $P$  with rules defining IDB-predicates and a database  $D$  with facts for EDB-predicates be given. Let  $\mathcal{M}$  be the minimal Herbrand model of  $P \cup D$  (i.e. the set of all derivable facts). A computation-sequence is a sequence  $\mathcal{S}$  of facts for IDB-predicates.*

- *The computation is correct iff all facts in  $\mathcal{S}$  appear in  $\mathcal{M}$ .*
- *It is complete iff all facts for IDB-predicates in  $\mathcal{M}$  appear in  $\mathcal{S}$ .*

Any algorithm for bottom-up evaluation of a Datalog program computes the facts of the minimal model in some such sequence. The specific order of the facts can be very different, though, which also influences how long intermediate facts must be stored. However, any reasonable computation sequence must have the following property: All facts in the sequence must be derivable from previously derived facts or given database facts. We call such computation sequences causal.

**Definition 3 (Causal Computation Sequence).** *A computation sequence  $\mathcal{S} = F_1, F_2, \dots$  is causal iff for every fact  $F_i$ ,  $i = 1, 2, \dots$  there is a rule  $A \leftarrow B_1 \wedge \dots \wedge B_n$  in  $P$  and a ground substitution  $\theta$  for this rule such that*

- $A\theta = F_i$  and
- $\{B_1\theta, \dots, B_n\theta\} \subseteq D \cup \{F_1, \dots, F_{i-1}\}$ .

**Lemma 1.** *Every causal computation sequence is correct.*

The translation result of the “Push” method contains a procedure “`start()`” which is basically the main program. It ensures that the start rules are applied which then lead to further procedure calls. Start rules are easy to execute because they contain only EDB body literals. The extensions of these predicates (database relations) are given, so the query corresponding to the rule body can be executed and the calls for the head literal can be done.

**Definition 4 (Requirements for the `start()` procedure).** *Let a program  $P$  and database  $D$  be given. The procedure `start()` ensures that for every start rule  $A \leftarrow B_1 \wedge \dots \wedge B_n$  in  $P$  and every ground substitution  $\theta$  for this rule such that  $B_i\theta \in D$  for  $i = 1, \dots, n$  the procedure call corresponding to  $A\theta$  is executed, i.e. the computation sequence contains this fact (under the assumption that the sequence is finite).*

Besides the `start()` procedure, the translation result contains one procedure per predicate. It must ensure that when a fact  $p(c_1, \dots, c_n)$  is derived, all applicable rule instances are fired that contain  $p(c_1, \dots, c_n)$  in the body. For simple (linear) rules this is easy, because they contain only one IDB body literal, so the true instances of the other body literals can be looked up in the database. For instances of complex rules (containing more than one IDB body literal), we must look at the global computation sequence: We can only require that the rule instance is applied if the needed instances of the other IDB body literals have already been derived earlier in the sequence. So in the case of complex rules, we need temporary storage for previously derived instances of IDB body literals.

**Definition 5 (Requirements for procedures implementing predicates).** *Let a program  $P$  and database  $D$  be given. For every IDB-predicate  $p$  of arity  $n$  there is a procedure  $\mathbf{p}$  with  $n$  arguments that ensures the following condition for the computation sequence  $\mathcal{S} = F_1, F_2, \dots$ :*

*For each  $i = 1, 2, \dots$ , if  $F_i$  is the first occurrence of the fact (i.e. there is no  $F_j$  with  $j < i$  and  $F_j = F_i$ ), the following holds:*

- *For each rule  $A \leftarrow B_1 \wedge \dots \wedge B_m$  and each ground substitution  $\theta$  for that rule such that there is a  $k \in \{1, \dots, m\}$  with  $B_k\theta = F_i$  and*

$$\{B_1\theta, \dots, B_m\theta\} \subseteq \{F_1, \dots, F_i\} \cup D,$$

*$A\theta$  appears somewhere in the sequence, i.e. this fact also occurs in  $\mathcal{S}$  (under the assumption that the sequence is finite).*

*It is not required that the position of the fact is after  $i$  (e.g., if it was already derivable with another rule instance).*

This simply means that all rule instances that are applicable when  $F_i$  is derived for the first time, are eventually applied. If we can ensure this condition, the completeness easily follows:

**Theorem 1 (Completeness).** *Let a program  $P$  and database  $D$  be given. If a computation sequence  $\mathcal{S}$  satisfies the conditions of Definitions 4 and 5, and if it is finite, it is complete, i.e. contains all IDB-facts in the minimal model of  $P \cup D$ .*

After clarifying the general approach and the requirements for each procedure, let us look a bit closer at the code that is generated. The overall code structure can be visualized by means of a rule application graph:

**Definition 6 (Rule Application Graph).** Let a Datalog program  $P$  be given and let  $IDB(P)$  be the set of IDB-predicates of this program, i.e. the predicates appearing in rule heads. The rule application graph for  $P$  is a bipartite directed graph  $(\mathcal{V}, \mathcal{E})$  with

- $\mathcal{V} := \mathcal{V}_1 \cup \mathcal{V}_2$ , where
  - $\mathcal{V}_1 := IDB(P)$  are “predicate nodes”, and
  - $\mathcal{V}_2 := \{(A \leftarrow B_1 \wedge \dots \wedge B_m, i) \mid A \leftarrow B_1 \wedge \dots \wedge B_m \in P, \text{ and } B_i \text{ is a literal with IDB-predicate}\}$  are “rule activation nodes”.

(The “active literal”  $B_i$  will be shown underlined.)
- $\mathcal{E} := \mathcal{E}_1 \cup \mathcal{E}_2$  with
  - $\mathcal{E}_1 := \{(p, (A \leftarrow B_1 \wedge \dots \wedge B_m, i)) \mid p \text{ is the predicate of } B_i\}$ ,
  - $\mathcal{E}_2 := \{((A \leftarrow B_1 \wedge \dots \wedge B_m, i), p) \mid p \text{ is the predicate of } A\}$ .

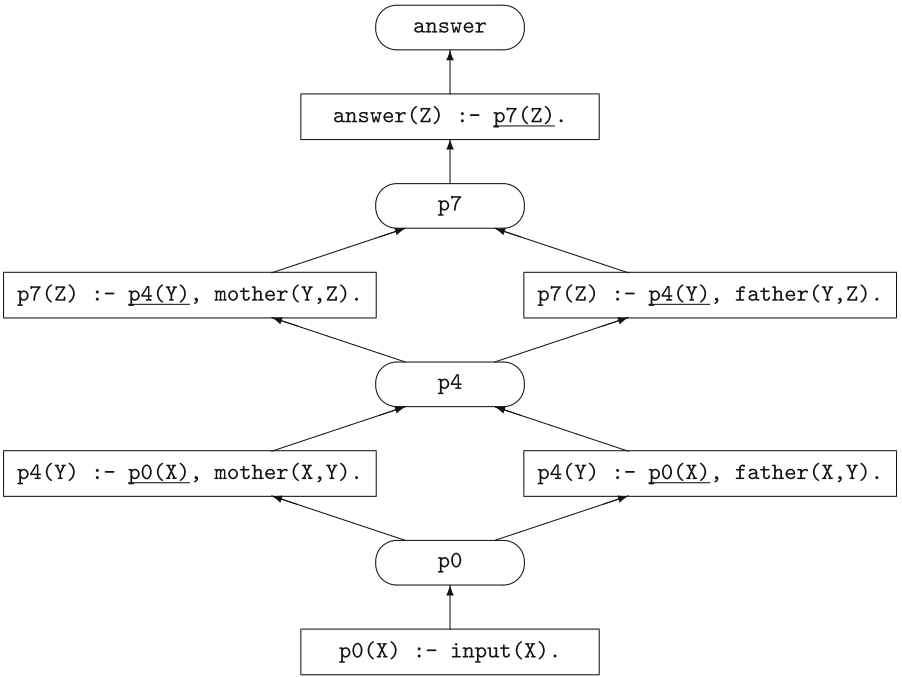


Fig. 1. Rule application graph for Example 1

The rule application graph for Example 1 is shown in Fig. 1. Each predicate node  $p$  corresponds to a procedure that contains a code block for each rule activation node that has an incoming edge from the predicate node. This code block ensures that the given fact for  $p$  is inserted for the active literal in that rule activation and the corresponding calls for the predicate in the head (to which

the outgoing edge leads) are done. Rule activation nodes without incoming edges correspond to start rules. They are executed in the procedure `start()` and initialize the stream of facts which is pushed along the edges through the nodes. In each rule activation node, an incoming fact can lead to multiple outgoing facts, but it can also be stopped if the actual data values do not match the active literal or if it does not find a join partner for the other body literals.

## 4.2 Example, Data Structures for Query Evaluation

The generated code for Example 1 is shown in Fig. 2. Loops over tuples returned from relation data structures are shown in pseudocode, one would use a cursor/iterator here.

```

void start()
{
    // p0(X) :- input(X).
    foreach X in input_f
        p0(X);
}

inline void p0(int c1)
{
    // p4(Y) :- p0(X), mother(X,Y).
    // X = c1
    foreach Y in mother_bf(c1)
        p4(Y);

    // p4(Y) :- p0(X), father(X,Y).
    // X = c1
    foreach Y in father_bf(c1)
        p4(Y);
}

inline void p4(int c1)
{
    // p7(Z) :- p4(Y), mother(Y,Z).
    // Y = c1;
    foreach Z in mother_bf(c1)
        p7(Z);

    // p7(Z) :- p4(Y), father(Y,Z).
    // Y = c1;
    foreach Z in father_bf(c1)
        p7(Z);
}

inline void p7(int c1)
{
    // answer(Z) :- p7(Z).
    // Z = c1;
    insert c1 into answer;
}

```

**Fig. 2.** Result of the push transformation for the program from Example 1

The data structures for the relations (EDB predicates) are as follows:

- `input_f` is a list of input values (probably only one). The suffix `f` is a binding pattern, it indicates that the relation is accessed with a free variable for the only column, i.e. the argument is an output argument. Our current implementation first loads all data to main memory, so a vector/list data structure is used in such cases, when only a full table scan is needed.
- `mother_bf` and `father_bf` support an access with a given (“bound”) first argument, where the second (“free”) argument is required. A map/multimap data structure is used in such cases.

When the data is loaded, we construct for each EDB body literal a relation data structure which just fits its use. I.e. if the literal contains constants or the same variable more than once, these selections are already evaluated while the data is loaded, and only matching tuples are stored. Variables which are bound when the literal is accessed form the search key of the index, i.e. we do an index join. Values for the other (“free”) variables are found with the index lookup. It might seem that we need a lot of memory for this, but benchmarks so far have shown that memory consumption is low compared to other systems and loading time is quick. Of course, one can construct examples where using such an optimal index for each body literal leads to a blowup. One should introduce a limit. It is always possible to simply load all data for an EDB predicate into a list, and then do the selection at runtime (which corresponds to a nested loop join).

For string data, we use a radix tree similar to the one in [9] to map each string to a unique, sequential integer. In this way, later comparisons can be done in one machine instruction, and index structures for relations are simpler and more efficient. It is usual also in Prolog systems to have a hash table for atoms (which are one type of strings). Therefore, the arguments and variables are shown with type `int` in the code. However, this is not essential for the Push method and will probably be relaxed in future versions of our implementation. Basically, one has to define the types for all columns of the database relations (EDB-predicates) and can then derive types of the IDB-predicate arguments.

### 4.3 Duplicate Elimination, Termination

We need to make sure that query evaluation terminates. Since there are only finitely many constants in the program and the database, the only reason for non-termination can be duplicates. In every recursive cycle, we must check for duplicate facts at least once. Even for non-recursive predicates, if many duplicates are generated for the predicate, it might greatly improve performance if these are detected early and further computations with them are avoided.

Of course, there is also a price to pay in form of a hash table or similar “set” data structure in which tuples must be inserted. Since the push method intends to avoid materialization of tuples, this is not nice. Note, however, that a data structure only for the purpose of duplicate elimination is still smaller than storing a general relation with arbitrary access. Of course, if we should know that tuples arrive in a sort order, duplicate elimination would be much cheaper. Since the variables which make up a tuple are assigned to at different frequencies (e.g., one in an outer loop, and one in an inner loop), we also intend to experiment with nested hash tables.

The user of our system can select for which predicates duplicate elimination is done (of course, recursive cycles must be broken). This is similar to switching tabling on for a predicate in a logic programming system that uses tabling.

If duplicate elimination is selected for a predicate  $p$ , the first thing the procedure for  $p$  does is to insert the argument tuple into a “set” data structure (e.g., a hash table). If this fails (because the tuple is already contained in the set), the procedure immediately returns.

#### 4.4 Temporary Relations for Complex Rules

The Push method tries to use each fact immediately when it is derived. This works nicely with simple rules which have only a single IDB body literal, and therefore all other facts needed to apply the rule are available in the database.

For complex rules (with more than one IDB body literal) it is unavoidable to keep derived facts matching a body literal until we have the matching facts (join partners) for the other IDB body literals. A rule instance is applied as soon as the last IDB fact used in this instance is derived (which is again the earliest possible time).

Thus, for complex rules, we create temporary relations for each IDB body literal. These relations contain all facts matching the respective body literal which have previously been derived. The code ensures the following condition: All ground instances of the rule where the instance of each body literal is contained in its temporary relation have already been applied, or are currently being applied (somewhere above in the recursion). Now, if for one of the body literals a new instance is derived, only this new instance is joined with all facts in the temporary relations for the other body literals. This new instance is also inserted into the temporary relation for its own body literal.

This also works if the same fact is used for two body literals: It is tried first for one of them, and then for the other. When it is tried for the first, the join partner is still missing, but it is inserted into its temporary relation. When it is matched with the second body literal, it is already stored in the relation for the first one, so it finds its join partner, and the rule instance is applied. Basically, this is seminaïve evaluation, where the “delta” consists of a single fact.

Note again that complex rules are an exceptional case in the output of the SLDMagic transformation. If the program was not generated by SLDMagic, one might eliminate some complex rules by unfolding. Furthermore, the above description works already in the most general case, where all IDB body literals are potentially recursive. In other cases, when we know that facts are first produced for one body literal, and then for the other one, we might eliminate some temporary relations. See Subject. 4.6.

*Example 2.* As an example of a complex and recursive rule, consider this version of the transitive closure:

```
tc(X, Y) :- edge(X, Y).
tc(X, Z) :- tc(X, Y), tc(Y, Z).
```

The corresponding code is shown in Fig. 3. Note that when a `foreach`-loop starts, the set of values over which it iterates must be fixed (insertions into the temporary relation have no effect on already active iterators).

```

void start()
{
    // tc(X, Y) :- edge(X, Y).
    foreach (X,Y) in edge_ff
        tc(X, Y);
}

void tc(int c1, int c2)
{
    // Duplicate check:
    if(!tc_bb.insert(c1, c2))
        return;

    // tc(X,Z) :- tc(X,Y), tc(Y,Z).
    // X=c1, Y=c2 (first body literal)
    tc_1_fb.insert(c1, c2);
    foreach Z in tc_2_bf(c2)
        tc(c1, Z);

    // tc(X,Z) :- tc(X,Y), tc(Y,Z).
    // Y=c1, Z=c2 (second body literal)
    tc_2_bf.insert(c1, c2);
    foreach X in tc_1_fb(c1)
        tc(X, c2);
}
    
```

**Fig. 3.** Result of the push transformation for the program from Example 2

#### 4.5 Code Block for a Rule Activation

Now we are ready to look at the code block for a rule activation

$$A \leftarrow B_1 \wedge \dots \wedge \underline{B_i} \wedge \dots \wedge B_m$$

in a bit more detail. It is structured as follows:

- It first has to check whether the given fact  $p(c_1, \dots, c_n)$  really matches the IDB body literal  $B_i = p(t_1, \dots, t_n)$ , i.e. for  $j = 1, \dots, n$ 
  - if  $t_j$  is a constant, then  $t_j = c_j$  must hold,
  - if  $t_j$  is a variable that appeared first in  $t_k$ ,  $k < j$ , then  $c_j = c_k$  must be satisfied (otherwise, the following code block is skipped).
- Note that if the procedure call contains constants as arguments, or the same variable in different arguments, the compiler might be able to evaluate the condition, and possibly remove the code block.
- If the rule is a complex rule,  $(c_1, \dots, c_n)$  is inserted into the temporary relation for  $B_i$ . (One can improve this by storing only values of variables that are later needed.)
- Then one executes the query  $B_1 \wedge \dots \wedge B_{i-1} \wedge B_{i+1} \wedge \dots \wedge B_m$  with the given values for the variables among the  $t_1, \dots, t_n$ . For simple rules, all these literals are EDB literals, so there is a given database relation for them. For complex rules, we created temporary relations for each IDB body literal in them, which are used here. So in both cases, relations are given for all  $B_j$ ,  $j \neq i$ , and it is a standard task to evaluate this query.
- The query result contains bindings for the remaining variables of the query. So we now have a ground substitution  $\theta$  for the rule such that  $(B_1 \wedge \dots \wedge B_m)\theta$  is true. With these variable values, the corresponding instance of the head literal  $A$  can be determined. Then the corresponding procedure call is done.



## 4.6 An Optimization for Complex Rules

Consider a rule with two IDB body literals, e.g.  $p(X)$  and  $q(X)$ . When a start rule is executed, only a subset of the nodes in the rule application graph are reachable from this start rule, i.e. only for some predicates facts are derived. If the first start rule yields only  $p$ -facts, but no  $q$ -facts, it is clear that the temporary relation for the second body literal is still empty. So the derived  $p$ -facts will be stored in the temporary relation for the first body literal, but cannot be pushed further. The corresponding code can be removed.

If for a later start rule  $p$ -facts are derived when  $q$ -facts have been derived in the meantime, this code is needed. Thus two different versions of the procedure are needed, which are used in the derivation for different start rules (also predicate-procedures between the start rule and this rule must be duplicated).

Another interesting special case is when all start rules that yield  $p$ -facts are executed before the first start rule with yields  $q$ -facts. In this case, the temporary relation for the body literal  $q(X)$  is not needed, and its facts can be simply pushed through the rule as if  $p$  were an EDB-predicate.

So in general, we have a sequence of components of the program, one for each start rule. For this optimization of the complex rules evaluation, it is be useful to distinguish three possible status values of a predicate  $p$  in component  $i$ :

- The predicate extension is empty, i.e. the predicate node for  $p$  is not reachable from any start rule  $\leq i$ .
- The predicate extension is partial, i.e.  $p$  is reachable in a component  $\leq i$  and  $p$  is also reachable in a component  $\geq i$ .
- The predicate extension is complete, i.e.  $p$  is not reachable in components  $\geq i$ .

## 4.7 Remarks About Inlining

Of course, unfolding procedure calls through inlining often leads to an increased code size, which might have negative effects on cache utilization and thus runtime. Currently, we leave the decision which procedure calls are inlined to the compiler (it does a cost-benefit estimation). With an analysis of the rules, we might be able to do better in future and use `__attribute__((always_inline))` or `__forceinline` on selected procedures.

## 5 Benchmark Results

We implemented our transformation and a supporting C++ library and checked several benchmarks from the OpenRuleBench benchmark suite [11]. The current version of our method can handle only basic Datalog without negation and structured terms. So we could try only such benchmark problems. However, for these we get encouraging results, often significantly faster than systems which are well established in the community. We tried systems that were considered also in the original OpenRuleBench test, namely XSB [15], YAP [8], and DLV [10]. Note that XSB is called a deductive database in [15], and clearly beats older,

well-known deductive database prototypes like CORAL [14]. The DLV system is strong in answer set programming, but it only uses system modules necessary for a given problem [10]. So it is not unfair to use it in a comparison for simpler problems. Recently, the Soufflé system has been developed [16]. It is not advertised as a deductive database, but as a system for doing static program analysis using Datalog. Nevertheless, it can be used to execute the benchmarks, and does a compilation to C++ as we do, although the code structure is different (it is based on relations and relational algebra). The performance results on the benchmarks we tried are comparable to our system (we restricted Soufflé to one core, because our prototype cannot do parallel evaluation yet). The significantly lower memory consumption of Soufflé in the compiler version is probably due to the use of a Trie data structure for relations (something we should try, too).

Of course, the above systems (except Soufflé) have been developed for more than a decade, and offer many features which are needed for the practical development of large projects. In contrast, our implementation is just a first prototype in order to check the potential of our evaluation method.

For space reasons, we show here only the results of two benchmarks [11]: The standard tail-recursive transitive closure program (with a cyclic graph with 1000 nodes and 50 000 edges), and a join of five relations with 10 000 rows each.

System	Load	Execution	Total time	Factor	Memory
Push (Proc.)	0.005 s	1.039 s	1.044 s	1.0	31.246 MB
Push (Switch)	0.005 s	1.030 s	1.033 s	1.0	23.303 MB
Seminaïve	0.005 s	1.670 s	1.672 s	1.6	31.210 MB
XSB	0.247 s	4.740 s	5.090 s	4.9	135.925 MB
YAP	0.240 s	10.549 s	10.833 s	10.4	147.601 MB
DLV	(0.373 s)	—	52.300 s	50.1	513.753 MB
Soufflé (SQLite)	(0.113 s)	—	11.237 s	10.8	43.083 MB
Soufflé (compiled)	(0.030 s)	—	0.790 s	0.8	3.810 MB

Transitive Closure Benchmark with query `tc(-,-)` [11]

System	Load	Execution	Total Time	Factor	Memory
Push (Proc.)	0.004 s	1.043 s	1.043 s	1.0	16.863 MB
Push (Switch)	0.004 s	1.031 s	1.032 s	1.0	9.010 MB
XSB	0.128 s	6.056 s	6.460 s	6.2	127.259 MB
YAP	0.207 s	3.572 s	3.840 s	3.7	135.921 MB
DLV	(0.253 s)	—	80.237 s	76.9	603.141 MB
Soufflé (SQLite)	(0.100 s)	—	12.680 s	12.2	38.548 MB
Soufflé (compiled)	(0.040 s)	—	1.450 s	1.4	4.264 MB

Join1 Benchmark with query `a(X,Y)` [11]

We compile to machine code, whereas XSB, YAP and DLV emulate an abstract machine. This explains a factor of about 3, see, e.g. [7]. In case of Soufflé, the compiled version uses an own implementation, whereas the interpreted version uses SQLite. Thus, the factor of 10 is not only caused by compilation.

Our transformation and the compilation together take about 0.5 s, which are not included in the above numbers (because they are done only once).

The additional memory in the procedure version of the Push method is required because the query result needs to be stored, whereas the original version could offer an iterator interface without having to store answer tuples.

## 6 Related Work

The general idea of immediately using derived facts to derive more facts is not new. For instance, variants of semi-naive evaluation have been studied which work in this way [17, 18]. Heribert Schütz proposed in his 1993 PhD thesis already a version of bottom-up evaluation that used procedures for each predicate called when a new fact was derived. In contrast to our approach, the rules were first normalized, and EDB predicates were not treated specially. There are differences also in the data structures and the structure of the generated code (our goal is doing partial evaluation). Furthermore, there was only a prototypical implementation in Lisp (as part of the LOLA system), and benchmark comparisons with established systems were not done or did not give the desired results.

The method is also related to the propagation of updates to materialized views. In [12], compiling Datalog rules to program code was studied based on incremental computation of the derived facts.

“Pushing” tuples through relational algebra expressions has also become an attractive technique for standard databases [13].

## 7 Conclusion

In this paper, we have explained a new version of the Push method for bottom-up evaluation in deductive databases. The main improvement is that the method is much simpler to understand, and the program code resulting from the Datalog-to-C++ transformation is much better structured. Current C++ compilers are able to do many optimizations automatically that were explicitly built into the old transformation. Our performance measurements on some benchmarks of the OpenRuleBench collection are encouraging. It seems that we are able to be faster than some well-established systems in the area. The current state of the project is reported on the following web page:

<http://users.informatik.uni-halle.de/~brass/push/>

## References

1. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic sets and other strange ways to implement logic programs. In: Proceedings of the 5th ACM Symposium on Principles of Database Systems (PODS 1986), pp. 1–15. ACM Press (1986)
2. Brass, S.: SLDMagic—the real magic (with applications to web queries). In: Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 1063–1077. Springer, Heidelberg (2000). <https://doi.org/10.1007/3-540-44957-4.71>
3. Brass, S.: Implementation alternatives for bottom-up evaluation. In: Hermenegildo, M., Schaub, T. (eds.) Technical Communications of the 26th International Conference on Logic Programming (ICLP 2010), Leibniz International Proceedings in Informatics (LIPIcs), vol. 7, pp. 44–53. Schloss Dagstuhl (2010). <http://drops.dagstuhl.de/opus/volltexte/2010/2582>
4. Brass, S.: Order in Datalog with applications to declarative output. In: Barceló, P., Pichler, R. (eds.) Datalog 2.0 2012. LNCS, vol. 7494, pp. 56–67. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32925-8\\_7](https://doi.org/10.1007/978-3-642-32925-8_7)
5. Brass, S.: A framework for goal-directed query evaluation with negation. In: Calimeri, F., Ianni, G., Truszczynski, M. (eds.) LPNMR 2015. LNCS (LNAI), vol. 9345, pp. 151–157. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-23264-5\\_14](https://doi.org/10.1007/978-3-319-23264-5_14)
6. Brass, S., Stephan, H.: Bottom-up evaluation of Datalog: preliminary report. In: Schwarz, S., Hölldobler, S. (eds.) 29th Workshop on (Constraint) Logic Programming (WLP 2015), pp. 21–35. HTWK Leipzig (2015). <http://www.imn.htwk-leipzig.de/WLP2015/>
7. Costa, V.S.: Optimising bytecode emulation for Prolog. In: Nadathur, G. (ed.) PPDP 1999. LNCS, vol. 1702, pp. 261–277. Springer, Heidelberg (1999). [https://doi.org/10.1007/10704567\\_16](https://doi.org/10.1007/10704567_16)
8. Costa, V.S., Rocha, R., Damas, L.: The YAP Prolog system. Theory Pract. Logic Programm. **12**(1–2), 5–34 (2012). <https://www.dcc.fc.up.pt/~ricroc/homepage/publications/2012-TPLP.pdf>
9. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013), pp. 38–49. IEEE Computer Society (1997). <http://www3.informatik.tu-muenchen.de/~leis/papers/ART.pdf>
10. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Trans. Comput. Logic **7**(3), 499–562 (2006). <https://arxiv.org/pdf/cs/0211004>
11. Liang, S., Fodor, P., Wan, H., Kifer, M.: OpenRuleBench: an analysis of the performance of rule engines. In: Proceedings of the 18th International Conference on World Wide Web (WWW 2009), pp. 601–610. ACM (2009). <http://rulebench.projects.semwebcentral.org/>
12. Liu, Y.A., Stoller, S.D.: From Datalog rules to efficient programs with time and space guarantees. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2003), pp. 172–183. ACM (2003). <http://www3.cs.stonybrook.edu/~liu/papers/Rules-PPDP03.pdf>
13. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. **4**(9), 539–550 (2011). <http://www.vldb.org/pvldb/vol4/p539-neumann.pdf>

14. Ramakrishnan, R., Srivastava, D., Sudarshan, S., Seshadri, P.: The CORAL deductive system. *VLDB J.* **3**, 161–210 (1994)
15. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: Snodgrass, R.T., Winslett, M. (eds.) *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD 1994)*, pp. 442–453 (1994). <http://user.it.uu.se/~kostis/Papers/xsbddb.html>
16. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in Datalog. In: *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*, pp. 196–206. ACM (2016)
17. Schütz, H.: *Tupelweise Bottom-up-Auswertung von Logikprogrammen (Tuple-wise bottom-up evaluation of logic programs)*. Ph.D. thesis, TU München (1993)
18. Smith, D.A., Utting, M.: Pseudo-naive evaluation. In: *Australasian Database Conference*, pp. 211–223 (1999). <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.177.5047>

# A Platform for Security Monitoring of Multi-cloud Applications

Pamela Carvalho<sup>1,2</sup>(✉), Ana R. Cavalli<sup>1,2</sup>, and Wissam Mallouli<sup>2</sup>

<sup>1</sup> SAMOVAR, Télécom SudParis, CNRS,  
Université Paris-Saclay, Évry, France

<sup>2</sup> Montimage, Paris, France

{pamela.carvalho,ana.cavalli,wissam.mallouli}@montimage.com

**Abstract.** This paper presents a security assurance platform to monitor and control the security in the context of multi-cloud applications. Indeed, this property is a crucial issue in multi cloud-based environments where many aspects need to be faced, including risk management, data privacy and isolation, security-by-design applications, and vulnerability scans. Moreover, it also becomes necessary to have an efficient system that interrelates and operates all security controls that are configured and executed independently on each component of the system.

In addition, as new attacks emerge every day, threat detection systems play a fundamental role in security monitoring schemes, identifying possible attacks. These systems handle an enormous volume of data, as they detect unknown malware by monitoring different activities from different points of observation, as well as adapting to new attack strategies and considering techniques to detect malicious behaviors and react accordingly.

In this paper, we describe a monitoring platform for securing multi-cloud applications, from a Service Level Agreement perspective. Moreover, we present a case study depicting the multi-cloud monitoring of a smart-city transport application for the citizens of Tampere, Finland. Considering the nature of the application under study, the service requires continuous execution and availability functionalities, as end-users may utilize the service at any time.

**Keywords:** Cloud computing · Security monitoring  
Service Level Agreement · Threat detection · Reaction

## 1 Introduction

Monitoring is a solution that is required to control the correct operation of the whole system running in a multi-cloud environment. According to the taxonomy proposed by [13, 14], the term multi-cloud denotes situations where a consumer (human or service) uses multiple, independent clouds, unlike to Cloud Federations that are achieved when a set of cloud providers voluntarily interconnect their infrastructures to allow sharing of resources among them. According to

the state of the art, few concrete multi-cloud solutions exist, topics addressed in research projects like MUSA, OPTIMIS, mOSAIC, MODAClouds, PaaSage and Cloud4SOA [6,12]. It is out of the scope of this paper to offer a complete survey of such activities. We suggest the interested reader the following works: [5,14,20].

Malfunctioning or even minor problems in a Virtual Machine (VM) could introduce vulnerabilities and stability issues to other VMs, as well as threaten the integrity of the host machine. In this paper, the monitoring function is needed to be able to precisely understand what is happening at network, system and application levels, pursuing a twofold objective. First, a proper monitoring system that improves the security in the communications and services offered by the multi-cloud virtual environments. Second, from the administration and management's point of view, a system that helps to ensure the environment's health, guarantee that the system works as expected and respects its Security Service Level Agreements (SecSLAs) [8].

In this context, we present a platform for monitoring multi-cloud based applications where each application component can be deployed in a different Cloud Service Provider (CSP). The proposed platform architecture raises several challenges when fulfilling an end-to-end security monitoring of the application execution and communication at runtime. These efforts are due to the complexity of cloud platforms that may consist of multiple layers and service paradigms (SaaS, PaaS, IaaS) and therefore need a flexible monitoring management in a distributed scheme.

To the best of our knowledge, no security monitoring solution has been designed for such multi-cloud distributed systems. Consequently, the main contribution of this paper is the design and deployment of a security assurance platform that gives an answer to these challenges along with preliminary results of a smart-city case study that provides efficient and optimal transportation to the half-a-million citizens of Tampere, Finland. This paper extends our work in progress paper [7], by presenting results of the performed experiments.

The paper is organized as follows. In Sect. 2, we present an overview of the multi-cloud security assurance platform and describe each of its modules. Section 3 presents the workflow for an use-case in this platform. Section 4 presents the related work on monitoring tools and threat detection systems. Section 5 gives some elements for discussion of the exposed work and presents the conclusion and future work.

## 2 The MUSA Security Assurance Platform SaaS

The MUSA Security Assurance Platform (MSAP) is part of the MUSA project framework, and is offered following the Software-as-a-Service (SaaS) model to the Cloud Service Client (CSC). The MSAP ensures the security of the whole application distributed across heterogeneous cloud providers. This platform integrates and offers the MUSA monitoring service, the MUSA enforcement support service and the MUSA notification service, all of them working together with

embedded security libraries. The monitoring service aims at evaluating the security and functional measures gathered by the use of multiple mechanisms such as standard APIs offered by the cloud provider or the security libraries. Furthermore, the monitoring service is able to trigger security alerts based on the event rules defined by the *Application DevOps team* (Fig. 1) following a SLA perspective. The notification service is in charge of sending the alerts to the CSC when relevant security incidents have been detected, so the *Application DevOps team* can react and adapt the application or the provisioned cloud resources if needed. At the same time, the enforcement support service collaborates with the MUSA security libraries to enforce the security protection of multi-cloud application components.

### 2.1 The MUSA Framework

The MUSA Framework relies on the MUSA H2020 project [1]. The main goal of this framework is to support the security-intelligent life-cycle management of distributed applications over heterogeneous cloud resources, through a security framework that includes: (a) security-by-design mechanisms to allow application self-protection at runtime, and (b) a reactive security approach, monitoring application at runtime to mitigate security incidents, so multi-cloud application providers can be informed and react to them without losing end-user trust in the multi-cloud application.

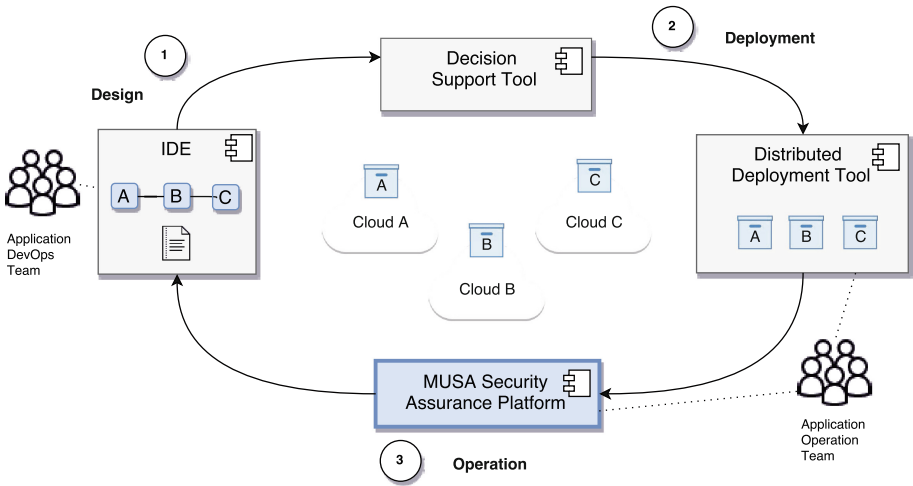


Fig. 1. MUSA Framework workflow

The MUSA framework workflow is depicted in Fig. 1. in three phases, where each element of the global architecture of the system is presented. The workflow begins when a CSC’s *Application DevOps team* uses the *IDE* module to specify



and design the multi-cloud application based on modeling techniques, taking into account (a) security requirements such as security embedded libraries for security at runtime, as well as (b) functional and business needs by delivering a composition of SLAs [8,9] with respect to each cloud component. The second phase is provided by the *Decision support tool* regarding cloud resource modeling through a continuous discovery and categorization of cloud services from different CSPs. Moreover, this module assists the *Application DevOps team* by selecting the set of combinations of cloud resources that best matches with the multi-cloud application functional and security needs. The third phase is the monitoring and operational stage, provided by the MSAP.

## 2.2 The MSAP Inputs

As mentioned previously, the MSAP fits the operation phase of the MUSA framework and considers two main inputs from the previous modules, in order to work properly:

- The Security SLA of the application to monitor: The MSAP recuperates the single application components SLAs or the multi-cloud composite application SLA. From single SLAs, the MSAP can monitor the security of single components, and from composite SLAs it can check the end-to-end security of the multi-cloud application taking the communication exchanges between remote components into account.
- The application deployment plan: From this plan, the MSAP recuperates the list of monitoring agents deployed with each application component as well as their IP addresses. This information is vital to link the monitoring agent with the application component to monitor the right security metrics that are specified in the application component security SLA.

The MUSA workflow is composed, as demonstrated in Fig. 2, of four main steps that come after gathering and preprocessing data from different monitoring agents. More details about these steps are provided in the next subsections.

## 2.3 Monitoring Agents

**Network Monitoring Agent.** Monitors a set of combined functionalities presented in the following list: (a) Data capture, filtering and storage (b) Events extraction and statistics collection, and (c) Traffic analysis and reporting providing, network, application, flow and user-level visibility.

This agent facilitates network performance monitoring and operation troubleshooting through its real-time and historical data gathering. With its advanced rules engine, the monitoring agent can correlate network events to detect performance, operational, and security incidents.

**System Monitoring Agent.** Monitors operating system resources which may be the cause of server performance degradation, and spots performance bottlenecks early on. The agent relies on Linux “top” command, which is frequently used by many system administrators to monitor Linux performance, being available in many Linux/Unix-like operating systems. The top command is used to display all the running and active real-time processes in an ordered list updating it regularly. It displays CPU usage, Memory usage, Swap Memory, Cache Size, Buffer Size, Process PID, User, among others.

**Application Monitoring Agent.** Monitors information about the internal state of the target system, i.e., multi-cloud application component to the MSAP during its operation. It notifies the MSAP about measurements of execution details and other internal conditions of the application component. The application monitoring agent is a Java library composed by two parts. The first is an aspect to be weaved into the application code via pointcuts in order to send application-internal tracing information to the MSAP for analysis. It is composed of a set of functions that can be weaved in strategic application points to capture relevant internal data. The second part connects the aspect with the notification tool via a connector library, providing a simple interface for sending log data to the MSAP in a secure way. In other words, the application monitoring agent is responsible for extracting the information from the system, and the connector is in charge of transferring it.

## 2.4 Preprocessing Module

This module has a particular challenge, which is extracting the right information from the collected data events provided by different monitoring agents and from different CSPs, in order to build the correct usage profiles. Additionally, in real cloud environments, periodic reports may be subject to loss or high latency, due to the applications elasticity or VM-related features (e.g., restarting a VM, rolling back). Therefore, this unit is meant to be dynamic, where features are analyzed regarding time-based contextual information. This has the advantage of decreasing the usage of resources for the analysis of large amounts of data, therefore increasing the performance of the framework and reasoning detection. Also following this direction, it is relevant at the moment of keeping a non-redundant knowledge and behavior dataset.

## 2.5 Metrics and Threat Analyzer

The detection module consists of two sub-modules: a Rule-based inspector and a Behavior profiler, as shown in Fig. 2.

The first relies on an engine that receives information events from the pre-processing module, regarding user’s access to non-authorized data, which are checked against these permission rules. Additionally to this policy control, some

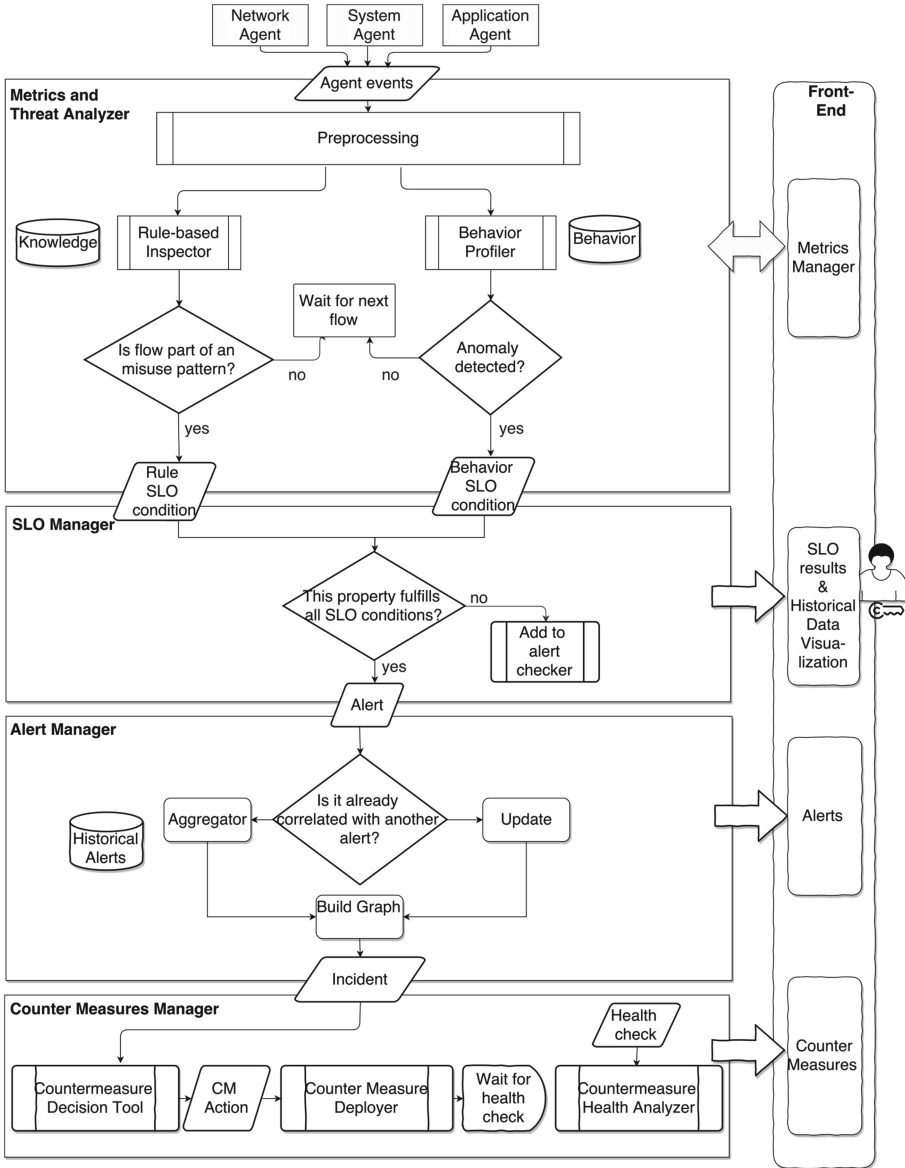


Fig. 2. A MSAP as a service instance general workflow

of the attributes obtained from the agents are inspected for specific pattern-matching detection.

The second module also receives the preprocessed data and comprehends two functions: the online learning and anomalies detection.

Most of the literature related to anomaly detection establishes a separated two-stages process, where systems are trained with *normal* data for second-stage comparison with new incoming information. This idea lacks dynamism, as cloud behavior may vary in mid-long term, and is highly dependent upon the nature of the training data. Therefore, the proposed self-learning module is capable of feeding and updating itself dynamically from new incoming data flows. This system will discriminate if it is appropriate to feed itself or not, lowering the possibility of training the engine with *malicious activity* as *normal*.

The model uses a semi-supervised learning, given the fact that new input data has not been labeled yet and it needs to be classified on the basis of their statistical properties only. The supervised component comprehends a smaller labeled dataset created in a lab environment, which learns from known attacks.

## 2.6 Service Level Objectives (SLO) Manager

The SLO Manager is able to check measured metrics to assert which objectives are useful in defining an anomalous behavior or a disrespected rule. The latter is already paved since it consists of rules that are continuously checked, but the challenge is designing criteria for stating SLO's for abnormal activity.

## 2.7 Alert Manager and Countermeasures Manager

The agents implement a prevention and mitigation methodology through a set of defenses, practices, and configurations prior to any attack, with the aim of reducing the impact of such attack. These issues may be addressed by network security, data protection, virtualization or isolation of resources.

The Incident handler responds to a policy-based alert and countermeasures mechanisms, given the severity of the incident diagnosed. This corresponds to the *Alert Manager* and *Countermeasure Manager* components from Fig. 2. The last module is intended to advise the CSPs and may consist in notifying the administrator to roll back the composite application, replicating a database, upgrading passwords complexity, disabling a specific user, among others.

The latter presents a crucial challenge because sometimes CSPs are unaware precisely of the countermeasures to consider because there are no established relationships between cloud components and their dependencies. This can be solved by clarifying these relationships.

# 3 Case Study: Service Availability in Smart-City Application

We studied the MSAP in the context of a multi-cloud platform for a smart city application, as depicted in Fig. 3. The TSM application (which stands for Tampere Smart Mobility) with the exception of the mobile application, has an architecture which is distributed in nature. Thus, each of the TSM components are

decoupled and developed independently. This application utilizes resources, services and information from the FMI (Finnish Meteorological Institute), Google directions and the Tampere Intelligent Transport System and Services (ITS) platform. The general schema is composed of:

1. TSM Engine (TSM<sub>E</sub>): It is an orchestrator which receives requests from different TSM components, analyses the requests according to several mappings, determines the appropriate TSM component required for processing the request and forwards the request to it.
2. Component Journey Planner (MJP): It provides multi-modal and optimal journey options based on the specified departure and destination points. Journey options are provided for buses, cars, cycling and walking.
3. Component Consumption Estimation Calculator (CEC): It calculates the energy needed to complete every journey option specified in the TSM application via a mobile application. It provides this information in a user-friendly way such as: the amount chocolate bars that would be burnt if the user chooses to walk to his/her destination.
4. Component IDM: It handles user authentication and authorization, as the security pillar of the entire TSM application. It authorizes the requests made over the resources and services that each TSM component exposes.

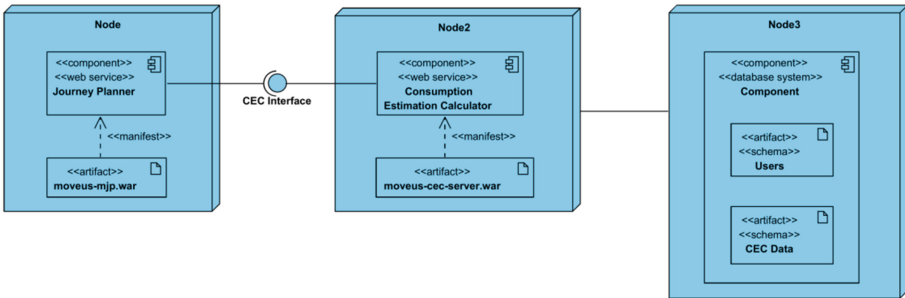


Fig. 3. Topology for multi-cloud case study

Given the real-time requirement for this platform, we decided to study the service availability metric, defined as a non-functional requirement, specified in terms of the percentage of time a system or a service is accessible [16]. To monitor service availability, we deployed dockers for each of the mentioned components and implemented the topology in OpenStack. We collected events from the system, network and application agents, and checked that the application processes are operative. Additionally, this metric uses the active monitoring module of the MSAP, in favor of checking the service availability from an end-user perspective. This observation is helpful given the fact that the agent events may show the service is running when it may not be visible from outside the cloud.

Continuing the MSAP flow, we parse the SLA and extract the SLO metric for service availability, as described in Fig. 4. This SLO is an individual example which is instantiated for each component.

---

```

<musa:SLO SLO_ID="1">
  <musa:Metric>Service availability</musa:Metric>
  <musa:SLOexpression>
    <musa:oneOpExpression operator="ge (>=)" operand="99.9"/>
  </musa:SLOexpression>
  <musa:importance_weight>HIGH</musa:importance_weight>
</musa:SLO>

```

---

**Fig. 4.** SLO XML for service availability metric in TSM application

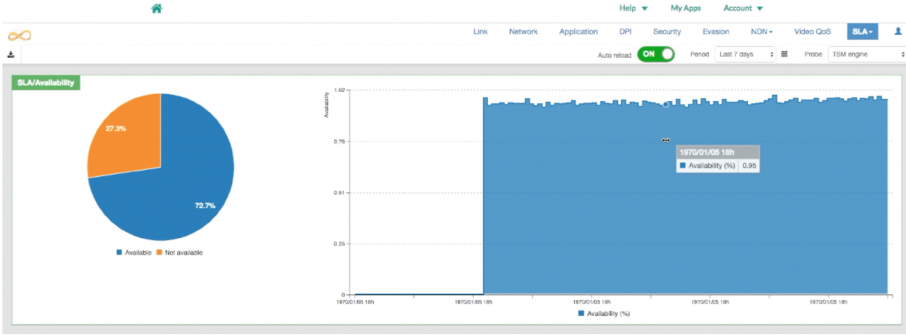
To assess the functionality of the MSAP, our testbed consisted in actively shutting down a component of the system and checking visualization results, alert notifications and countermeasures activations, through the dashboard front-end. Figure 5a. and b. correspond to the metrics presented in dashboard of the MSAP, and contain all the monitored historical data regarding the service availability metric. Detailed in Fig. 5a., the first white part of the time slot in the graph accurately presents the unavailability of service for several seconds (also depicted as the orange portion of the pie chart). Additionally, Fig. 5b. illustrates the numerous notifications for all the period of evaluation.

The Countermeasure Manager module of the MSAP is in charge of using a High Availability (HA) framework<sup>1</sup> as a way of reacting to a possible alert or violation of this metric. This HA framework, one of the security enforcement mechanisms of the MUSA framework, is based on an open-source software built around the Corosync/Pacemaker stack, patched and configured to work together to bring clustering mechanisms to multi-cloud-based services. This framework encapsulates each of the TSM application components (e.g., TSMc, CEC, IDM and MJP) and handles the task of their deployment in a redundant way managing thus different availability failures and proposing a fault tolerant system that guarantees the availability of different TSM components.

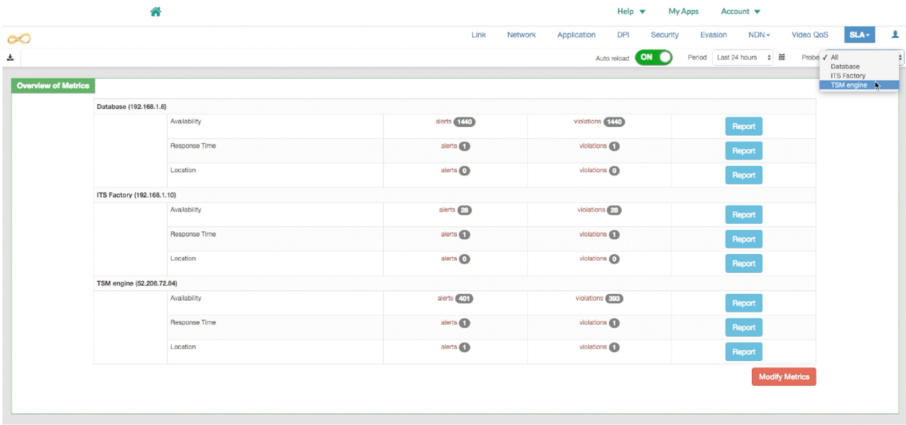
## 4 Related Work

From the monitoring perspective, current solutions to assess security can still be used in virtualized network environments [2,4]. Nevertheless, they need to be adapted and correctly controlled since they were meant mostly for physical and not virtual systems, and they do not allow fine-grained analysis tailored to the needs of CSCs and virtualized networks. The lack of visibility, controls on internal virtual networks and the heterogeneity of devices used, make many performance assessment applications ineffective. On one hand, the impact of virtualization on these technologies needs to be assessed. On the other hand, these

<sup>1</sup> <https://dSPACE.cc.tut.fi/dpub/handle/123456789/24492?locale-attribute=en>.



(a) Visualization



(b) Notifications

Fig. 5. MSAP dashboard

technologies need to cope with ever-changing contexts and trade-offs between the monitoring costs and benefits involved. Here, virtualization of application components facilitates changes, making it necessary for monitoring applications to keep up with this dynamic behavior.

Solutions such as Ceilometer [2], a monitoring solution for OpenStack, provide efficient collection of metering data regarding CPU and network costs. However, it is focused on creating a unique contact point for billing systems to acquire all of the measurements they need, and it is not oriented to perform any action to improve the metrics that it monitors. Furthermore, security issues are not part of the monitored features. StackTach [4] is another example oriented to monitor performance for billing purposes by auditing the OpenStack’s Nova component. Similarly, but not specifically oriented to billing, Collected [10] gathers system performance statistics and provides mechanisms to store the collected values. A recent project from OPNFV named Doctor [3], focuses on the creation of a

fault management and maintenance framework for high availability of network services on top of virtualized infrastructures.

In terms of security, OpenStack provides a security guide [15] with best practices determined by cloud operators when deploying their solutions. Some tools go deeper to guarantee certain security aspects in OpenStack, for instance: Bandit [18] provides a framework for performing security analysis of Python source code; Consul [11] is a monitoring tool oriented to service discovery that also performs health checking to prevent routing requests to unhealthy hosts.

Also, threat detection systems in cloud-based environments usually enhance security mechanisms by monitoring system's health. They correspond to a hardware device or software application that monitors activity (e.g., from network, VM host, user) for malicious policy violations. Zbakh et al. evaluated in [19] several Intrusions Detection Systems (IDS) architectures through proposed multi-criteria decision technique, according to the above-introduced requirement together with few others such as: Performance, availability, scalability, secure and encrypted communication channels, transparency with respect to end-users, information security policies as input to the architecture, accuracy including the number of false positives (FP) and false negatives (FN) and detection methods used, among others.

According to such literature, IDS architectures may vary if they are distributed, centralized, agent-based or collaborative [19]. Patel et al. [17] provided an extended systematic-based study of intrusion detection systems, presenting a classification with regards to response time, alarm management, detection method, data collection type, among others. In general, these systems are designed with the following modules: data capturing (Sect. 2.3) and preparation (Sect. 2.4), which function as an input for the data analysis and detection (Sect. 2.5). The latter functionality corresponds to the algorithms implemented to detect suspicious activities and known attack patterns.

## 5 Conclusion and Future Work

The MUSA Security Assurance Platform is proposed as a service that needs to be deployed in the suitable CSP (or CSPs since we can divide the platform into multiple components or micro-services). It offers a set security controls and requirements according to the application needs. Moreover, the MSAP is able to enforce the security of multi-cloud applications by executing the necessary countermeasures to security requirements or to mitigate undesired issues. Its real-time data collection and analysis, together with its virtualized (cloud-based) nature, makes the MSAP a powerful tool to provide multi-cloud applications with end-to-end assurance capabilities.

In detail, this platform presents several advantages, as includes techniques to perform the monitoring of applications that are deployed over heterogeneous cloud resources. It is also based in the concept of monitoring security metrics from SLAs to detect potential deviations and trigger countermeasures to protect applications against attacks and anomalies. This service is available following this link <http://assurance-platform.musa-project.eu/> and a demonstration



of the tool for the presented use-case is available on You-Tube following this link: <https://www.youtube.com/watch?v=zc6p-0H9yFo>.

As future work, we consider experimenting with an automatic deployment of reactive countermeasures. Additionally, we plan on extending the set of security metrics available for monitoring, by enhancing our monitoring agents and by developing new techniques for detection in the Metrics and Threat Analyzer module. This last section will focus further on the detection of unknown threats with anomaly-behavior detection techniques.

**Acknowledgement.** The work presented in this paper has been developed in the context of the MUSA EU Horizon 2020 project [1] under grant agreement No 644429.

## References

1. Musa project. <http://www.musa-project.eu/>. Accessed Jan 2017
2. Openstack ceilometer. <http://docs.openstack.org/developer/ceilometer/>. Accessed Jan 2017
3. Opnfv doctor. <http://wiki.opnfv.org/doctor>. Accessed Jan 2017
4. Stacktach. <http://stacktach.readthedocs.org/en/latest/index.html>. Accessed Jan 2017
5. Lifecycle management of service-based applications on multi-clouds: a research roadmap (2013)
6. Multi-Cloud: expectations and current approaches (2013)
7. Carvallo, P., Cavalli, A.R., Mallouli, W., Rios, E.: Multi-cloud applications security monitoring. In: Au, M.H.A., Castiglione, A., Choo, K.-K.R., Palmieri, F., Li, K.-C. (eds.) GPC 2017. LNCS, vol. 10232, pp. 748–758. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-57186-7\\_54](https://doi.org/10.1007/978-3-319-57186-7_54)
8. Casola, V., Benedictis, A.D., Modic, J., Rak, M., Villano, U.: Per-service security sla: A new model for security management in clouds. In: 2016 IEEE 25th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 83–88, June 2016
9. Casola, V., Benedictis, A.D., Rak, M., Rios, E.: Security-by-design in clouds: a security-sla driven methodology to build secure cloud applications. *Procedia Comput. Sci.* **97**, 53–62 (2016). <http://www.sciencedirect.com/science/article/pii/S1877050916320968>. 2nd International Conference on Cloud Forward: From Distributed to Complete Computing
10. Collectd. <http://collectd.org/>. Accessed Jan 2017
11. Consul. <https://www.consul.io/>. Accessed Jan 2017
12. Ferry, N., Rossini, A., Chauvel, F., Morin, B.: Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems. In: 2013 IEEE Sixth International Conference on Cloud Computing (2013)
13. Global Inter-cloud Technology Forum: Use Cases and Functional Requirements for Inter-Cloud Computing. Technical report (2010)
14. Grozev, N., Buyya, R.: Inter-Cloud architectures and application brokering: taxonomy and survey. *Softw. Pract. Exp.* **44**(3), 369–390 (2012)
15. OpenStack Security Guide. <http://docs.openstack.org/sec/>. Accessed Jan 2017
16. Nabi, M., Toeroe, M., Khendek, F.: Availability in the cloud: state of the art. *J. Netw. Comput. Appl.* **60**, 54–67 (2016)

17. Patel, A., Taghavi, M., Bakhtiyari, K., Celestino Júnior, J.: An intrusion detection and prevention system in cloud computing: a systematic review. *J. Netw. Comput. Appl.* **36**(1), 25–41 (2013)
18. Bandit Project. <http://wiki.openstack.org/wiki/Security/Projects/Bandit>. Accessed Jan 2017
19. Zbakh, M., Elmahdi, K., Cherkaoui, R., Enniari, S.: A multi-criteria analysis of intrusion detection architectures in cloud environments. In: 2015 International Conference on Cloud Technologies and Applications (CloudTech), pp. 1–9. IEEE (2015)
20. Zeginis, C., Kritikos, K., Garefalakis, P., Konsolaki, K., Magoutis, K., Plexousakis, D.: Towards cross-layer monitoring of multi-cloud service-based applications. In: Lau, K.-K., Lamersdorf, W., Pimentel, E. (eds.) ESOCC 2013. LNCS, vol. 8135, pp. 188–195. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40651-5\\_16](https://doi.org/10.1007/978-3-642-40651-5_16)

# The Hybrid Multidimensional-Ontological Data Model Based on Metagraph Approach

Valeriy M. Chernenkiy, Yuriy E. Gapanyuk<sup>(✉)</sup>, Anatoly N. Nardid,  
Anton V. Gushcha, and Yuriy S. Fedorenko

Informatics and Control Systems Department, Bauman Moscow State  
Technical University, Baumanskaya 2-ya, 5, 105005 Moscow, Russia  
{chernen, gapyu}@bmstu.ru, nazgull09@gmail.com,  
ncrashed@gmail.com, fedyura1992@yandex.ru

**Abstract.** This paper is aimed to overcome the limitation of the traditional multidimensional model in order to allow usage of numerical, textual and object-oriented information as multidimensional model measures. The ontological approach is reviewed. The formal definition of multidimensional approach is given. The idea of multidimensional and ontological approaches hybridization is discussed. The hybrid multidimensional-ontological data model requirements are proposed. The formal definitions of the metagraph data model and metagraph agent model are given. The examples of data metagraph and metagraph rule agent are discussed. The representation of object-oriented data structures in form of metagraph is given. The hybrid multidimensional-ontological data model based on metagraph approach is proposed. Predicate representation of metagraph model considered as a physical data model for metagraph approach implementation is given.

**Keywords:** Ontology · Multidimensional data model · Dimension Measure · Metagraph · Metagraph agent

## 1 Introduction

Traditionally, ontologies are used to describe complex knowledge domains, and multidimensional data model to build analytical systems based on numerical measures.

The multidimensional data model allows declaring the numerical measures that reside at the intersection of independent dimensions. But information became more and more complex and multidimensional data model numerical measures limitation became appreciable.

The graph-based analytical systems are developed for complex information analysis. There are a number of graph databases used as the basis for analytical systems such as Neo4j, ArangoDB, and OrientDB.

Nowadays it may be noted the tendency for complicating and hybridizing graph database data models [1, 2]. The example of this tendency is the HypergraphDB database that is the component of OpenCog AI project. As the name implies HypergraphDB uses complex hypergraph model as a data model. There is a GRAKN.AI project also

aimed for AI purpose that explicitly combines graph-based and ontology-based approach for data analysis.

Thus, the graph-based analytics explicitly or implicitly uses ontological approach advantages. But the advantages of multidimensional approach are lost in this case.

The purpose of this paper is to propose a hybrid model that combines advantages of multidimensional and ontological (graph-based) approaches. To achieve this purpose, we first consider ontological and multidimensional approaches separately and then hybridizing them using metagraph approach.

## 2 Ontological Approach

The classical Gruber ontology definition is “an explicit specification of a conceptualization” [3].

We will use ontology formalization proposed by Gavrilova and Chernigovskava [4]:

$$O = \langle T, \mathfrak{R}, \Phi \rangle, T = \{term_i\}, \mathfrak{R} = \{rel_j\},$$

where  $O$  – formal ontology;  $T$  – set of concepts (terms)  $term_i$  of ontology;  $\mathfrak{R}$  – set of semantic relations  $rel_j$  between concepts;  $\Phi$  – set of interpretation functions defined on the sets of concepts and relations of ontology.

If  $\mathfrak{R} = \{is\_a\}$ ,  $\Phi = \emptyset$ , then the ontology is a taxonomy of concepts. If  $\mathfrak{R} = \{part\_of\}$ ,  $\Phi = \emptyset$ , then the ontology is a meronomy or paronomy of concepts. If  $\mathfrak{R} = \{is\_a, part\_of\}$ ,  $\Phi = \emptyset$ , then the ontology is a metasystem of concepts.

There is a frequently used variation of the ontology definition which does not contain  $\Phi$  component the so-called lightweight ontology ( $OL$ ):

$$OL = \langle T, \mathfrak{R} \rangle \tag{1}$$

Today one of the most developed technologies for working with ontologies is the semantic web technology. In this case, Resource Description Framework (RDF) is used as a data model, SPARQL is used as a query language. RDFS and OWL (OWL2) are used as ontology definition languages on the base of RDF. Using RDFS and OWL it is possible to express various relationships between ontology elements (class, subclass, equivalent class, etc.) [5]. For RDF persisting and SPARQL processing special RDF storage systems are used e.g. Apache Jena, AllegroGraph.

Semantic web technologies are brought to the level of industrial technologies and are used in a number of information systems. But this approach has several limitations. The first limitation is that the RDF data model consists of very small data item – “subject-predicate-object” triples. As a result, an average sized relational database may correspond to a triple store containing billions of triples. The second limitation is the N-ary relation limitation [6]. This limitation is that the RDF model does not allow simple ways to describe N-ary relations between vertices of the semantic graph, which complicates the description of complex situations in the semantic graph.

### 3 Multidimensional Approach

Multidimensional data model, proposed by Codd et al., allows working with numerical data (measures) binding them to the hierarchical taxonomies (dimensions) [7]. The multidimensional data model is a core for OLAP (online analytical processing) information systems.

Many variants of formalization of such a model were proposed, for example [8]. In this section, we use our own simplified version of the formalization, which will help to describe the proposed hybrid model. A multidimensional hypercube may be described as follows:

$$\begin{aligned} HC &= \langle HCD, MSR, HCF, HCR \rangle, \\ HCD &= \{hcd_i\}, MSR = \{msr_i\}, HCF = \{hcf_j\}, HCR = \{hcr_k\}, \end{aligned} \quad (2)$$

where  $HC$  – hypercube;  $HCD$  – set of hypercube dimensions ( $hcd_i$  – dimension);  $MSR$  – set of hypercube measures ( $msr_i$  – measure);  $HCF$  – set of hypercube facts ( $hcf_i$  – fact);  $HCR$  – set of hypercube aggregation rules ( $hcr_i$  – rule).

Hypercube dimension:

$$hcd_i = \langle \{hcd_i^k\}, \prec \rangle,$$

where  $hcd_i^k$  – hypercube dimension element;  $\prec$  – a partial order on the set of hypercube dimension elements.

In most cases, hypercube dimension element is organized in a tree structure, in case of time dimension e.g. year  $\rightarrow$  month  $\rightarrow$  week  $\rightarrow$  day. But partial order organization is correct than tree structure organization because partial order organization allows describing ragged hierarchies, in case of time dimension e.g. the month  $\rightarrow$  week  $\rightarrow$  day and month  $\rightarrow$  decade  $\rightarrow$  day hierarchies are allowed to exist simultaneously in one dimension.

Using partially ordered set for dimension definition corresponds to the traditional approach. But if we care about semantic relations between dimension elements it is more convenient to represent the dimension in the form of lightweight ontology:

$$hcd_i \equiv OL \quad (3)$$

In this case, dimension elements correspond to the concepts of the ontology. But instead of non-semantic partial order on the set of dimension elements, there is a set of semantic relations is used allowing representing various semantic relations between dimension elements. Using semantic relations either tree structure organization or partial order organization may be successfully emulated.

Hypercube fact:

$$hcf_j = \langle \{hcd_i^{ref}\}, \{msr_n\} \rangle, \quad (4)$$

where  $hcd_i^{ref}$  – reference to the dimension element;  $msr_n$  – measure.

In case of low-level hypercube fact, the set  $\{hcd_i^{ref}\}$  contains references to low-level elements of all hypercube dimensions.

In case of aggregated hypercube fact  $\{hcd_i^{ref}\} \in P(HCD)$ , the set  $\{hcd_i^{ref}\}$  belongs to the powerset of all hypercube dimensions because aggregation rules may exclude dimensions during the aggregation process. Simultaneously, during aggregation, elements  $hcd_i^{ref}$  roll up upon their hierarchies, providing data aggregation on higher levels of hierarchies.

Hypercube aggregation rule:

$$hcr_k : \{hcf_{OUT}\} = agf(\{hcf_{IN}\}, HCD^{ag}), HCD^{ag} \subset HCD, \tag{5}$$

where  $hcf_{OUT}$  – output (aggregated) facts;  $agf$  – aggregation function;  $hcf_{IN}$  – input (non-aggregated) facts;  $HCD^{ag}$  – subset of hypercube dimensions used in aggregation.

Aggregation rules allow calculating aggregated facts on the base of non-aggregated or low-level aggregated facts and hypercube dimensions. The typical aggregation functions are *count*, *sum*, *min*, *max* and other numerical functions.

Depending on multidimensional system realization aggregation rules may be bound to the particular dimensions or to the whole hypercube.

Today multidimensional model is used in a great number of information systems. The advantages of the multidimensional model are worldwide recognized. But the multidimensional model is oriented for numerical measures usage. Textual or object-oriented information are not considered for use as measures. This may be noted as a limitation of the multidimensional model.

## 4 The Idea of Multidimensional and Ontological Approaches Hybridization

Let us consider some of the considerations that are significant to the multidimensional and ontological data models hybridization.

Usually, in the multidimensional data model, the most hypercube dimensions are ontologies-meronomies. In the given example of time dimension “year → month → week → day” the relation “→” actually means *part\_of* relation. It gives the idea that ontology is well suited for separate dimension description but doesn’t fit well for the description of the entire hypercube.

The significant idea of the multidimensional data model is that it allows declaring the data that reside at the intersection of independent dimensions. If we want to extend the scope of the information system, we add new dimensions to our model, which leads to adding new dimension references element to the set  $\{hcd_i^{ref}\}$  of hypercube facts definition. This makes multidimensional data model well suitable for extendable information systems definition. But the limitation of the multidimensional model is its orientation for numerical measures. Thus, the traditional multidimensional model is not suitable for textual or object-oriented measures.

The “Semantic OLAP” project [9] is a variant of hybridization of multidimensional and ontological approaches. Semantic Mediawiki system is an example of this approach implementation. In this case, RDF storage is used as a data source. Hypercube dimensions and hypercube facts are stored as RDF triples. SPARQL query language is used for data retrieval and aggregation. This approach should be noted as an innovative. The idea of a homogeneous storage model for dimensions and facts in form of RDF triples is very interesting. But this approach does not overcome the main limitation of traditional multidimensional data model because measures in this approach are only numerical.

In view of the above considerations let us formulate the “hybrid multidimensional-ontological data model requirements”:

1. The hybrid model allows describing hypercube dimensions in form of ontology.
2. Numerical, textual information and object-oriented data structures are considered for use as measures.
3. Proposed model should provide aggregation capabilities comparable to the aggregation function in the traditional multidimensional model.

To meet these requirements, we need the approach that allows describing ontological and multidimensional models and provides aggregation capabilities. We propose to use a metagraph approach for these purposes.

## 5 The Metagraph Model Definition

Metagraph is a kind of complex network model, proposed by Basu and Blanning in their book [10] and then adapted for information systems description in our paper [11]. According to [11]:

$$MG = \langle V, MV, E \rangle, \quad (6)$$

where  $MG$  – metagraph;  $V$  – set of metagraph vertices;  $MV$  – set of metagraph metavertrices;  $E$  – set of metagraph edges.

Metagraph vertex is described by a set of attributes:

$$v_i = \{atr_k\}, v_i \in V, \quad (7)$$

where  $v_i$  – metagraph vertex;  $atr_k$  – attribute.

Metagraph edge is described by a set of attributes, the source and destination vertices and edge direction flag:

$$e_i = \langle v_S, v_E, eo, \{atr_k\} \rangle, e_i \in E, eo = true \mid false, \quad (8)$$

where  $e_i$  – metagraph edge;  $v_S$  – source vertex (metavertex) of the edge;  $v_E$  – destination vertex (metavertex) of the edge;  $eo$  – edge direction flag ( $eo = true$  – directed edge,  $eo = false$  – undirected edge);  $atr_k$  – attribute.

The metagraph fragment:

$$MG_i = \{ev_j\}, ev_j \in (V \cup E \cup MV), \quad (9)$$

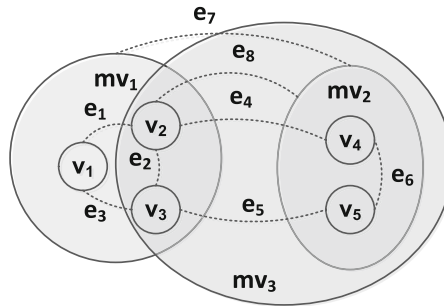
where  $MG_i$  – metagraph fragment;  $ev_j$  – an element that belongs to the union of vertices, edges, and metaverices.

The metagraph metavertex:

$$mv_i = \langle \{atr_k\}, MG_j \rangle, mv_i \in MV, \quad (10)$$

where  $mv_i$  – metagraph metavertex belongs to set of metagraph metaverices  $MV$ ;  $atr_k$  – attribute,  $MG_j$  – metagraph fragment.

Thus, metavertex in addition to the attributes includes a fragment of the metagraph. The presence of private attributes and connections for metavertex is distinguishing feature of metagraph. It makes the definition of metagraph holonic – metavertex may include a number of lower level elements and in turn, may be included in a number of higher level elements.



**Fig. 1.** Example of data metagraph

The example of data metagraph (shown at Fig. 1) contains three metaverices:  $mv_1$ ,  $mv_2$  and  $mv_3$  (emphasized with gray background). Metavertex  $mv_1$  contains vertices  $v_1, v_2, v_3$  and connecting them edges  $e_1, e_2, e_3$ . Metavertex  $mv_2$  contains vertices  $v_4, v_5$  and connecting them edge  $e_6$ . Edges  $e_4, e_5$  are examples of edges connecting vertices  $v_2-v_4$  and  $v_3-v_5$  are contained in different metaverices  $mv_1$  and  $mv_2$ . Edge  $e_7$  is an example of the edge connecting metaverices  $mv_1$  and  $mv_2$ . Edge  $e_8$  is an example of the edge connecting vertex  $v_2$  and metavertex  $mv_2$ . Metavertex  $mv_3$  contains metavertex  $mv_2$ , vertices  $v_2, v_3$  and edge  $e_2$  from metavertex  $mv_1$  and also edges  $e_4, e_5, e_8$  showing holonic nature of the metagraph structure.



## 6 The Metagraph Agent Definition

The metagraph model is aimed for complex data description. But it is not aimed for data transformation. To solve this issue the metagraph agent ( $ag^{MG}$ ) aimed for data transformation is proposed. There are two kinds of metagraph agents: the metagraph function agent ( $ag^F$ ) and the metagraph rule agent ( $ag^R$ ). Thus  $ag^{MG} = ag^F \mid ag^R$ .

The metagraph function agent serves as a function with input and output parameter in form of metagraph:

$$ag^F = \langle MG_{IN}, MG_{OUT}, AST \rangle, \quad (11)$$

where  $ag^F$  – metagraph function agent;  $MG_{IN}$  – input parameter metagraph;  $MG_{OUT}$  – output parameter metagraph;  $AST$  – abstract syntax tree of metagraph function agent in form of metagraph.

The metagraph rule agent is rule-based:

$$ag^R = \langle MG, R, AG^{ST} \rangle, R = \{r_i\}, r_i : MG_j \rightarrow OP^{MG}, \quad (12)$$

where  $ag^R$  – metagraph rule agent;  $MG$  – working metagraph, a metagraph on the basis of which the rules of agent are performed;  $R$  – set of rules  $r_i$ ;  $AG^{ST}$  – start condition (metagraph fragment for start rule check or start rule);  $MG_j$  – a metagraph fragment on the basis of which the rule is performed;  $OP^{MG}$  – set of actions performed on metagraph.

The antecedent of the rule is a condition over metagraph fragment, the consequent of the rule is a set of actions performed on metagraph. Rules can be divided into open and closed.

The consequent of the open rule is not permitted to change metagraph fragment occurring in rule antecedent. In this case, the input and output metagraph fragments may be separated. The open rule is similar to the template that generates the output metagraph based on the input metagraph.

The consequent of the closed rule is permitted to change metagraph fragment occurring in rule antecedent. The metagraph fragment changing in rule consequent cause to trigger the antecedents of other rules bound to the same metagraph fragment. But incorrectly designed closed rules system can cause to an infinite loop of metagraph rule agent.

Thus, metagraph rule agent can generate the output metagraph based on the input metagraph (using open rules) or can modify the single metagraph (using closed rules).

The example of metagraph rule agent is shown in Fig. 2. The metagraph rule agent “metagraph rule agent 1” is represented as metagraph metavertex. According to the definition it is bound to the working metagraph  $MG_1$  – a metagraph on the basis of which the rules of the agent are performed. This binding is shown with edge  $e_4$ .

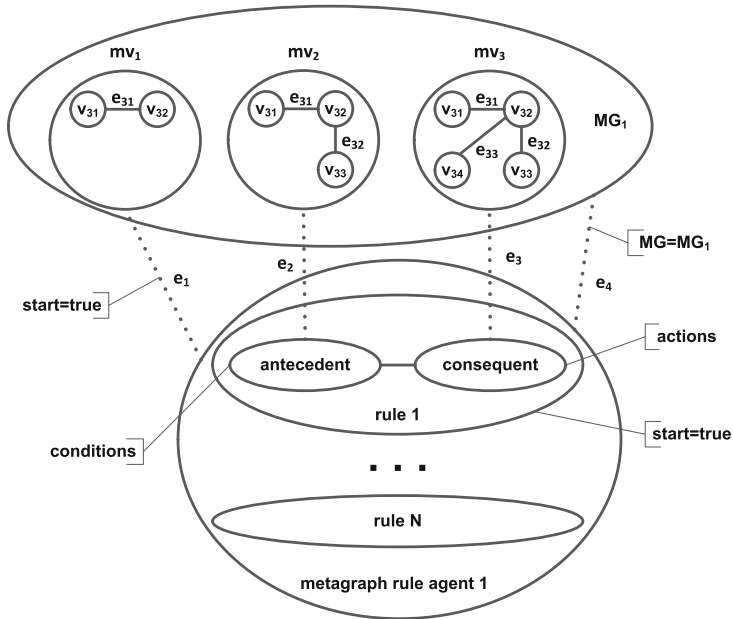


Fig. 2. Example of metagraph rule agent

The metagraph rule agent description contains inner metavertices corresponds to agent rules (rule 1 ... rule N). Each rule metavertex contains antecedent and consequent inner vertices. In given example  $mv_2$  metavertex bound with antecedent which is shown with edge  $e_2$  and  $mv_3$  metavertex bound with consequent which is shown with edge  $e_3$ . Antecedent conditions and consequent actions are defined in form of attributes bound to antecedent and consequent corresponding vertices.

The start condition is given in form of attribute “start = true”. If the start condition is defined as a start metagraph fragment then the edge bound start metagraph fragment to agent metavertex (edge  $e_1$  in given example) is annotated with attribute “start = true”. If the start condition is defined as a start rule than the rule metavertex is annotated with attribute “start = true” (rule 1 in given example). Figure 2 shows both cases corresponding to the start metagraph fragment and to the start rule.

The distinguishing feature of the metagraph agent is its homoiconicity which means that it can be a data structure for itself. This is due to the fact that according to definition metagraph agent may be represented as a set of metagraph fragments and this set can be combined in a single metagraph. Thus, the metagraph agent can change the structure of other metagraph agents.

## 7 The Representation of Object-Oriented Data Structures in Form of Metagraph

In this section, we consider the basics of the object-oriented data structures representation using metagraph approach. We review only data structures containing data fields in form *name:type:value* where type may be atomic type, complex type or list (collection) type. We suggest that this structure is enough complicated for representing basic analytical data. The representation of more complicated data structures (e.g. containing methods) requires a separate study and is out of the scope of this paper.

The data structure is defined as follows:

$$DS = \langle ds_T, DS_F \rangle, ds_T \in TP, DS_F = \{fld^i\}, \quad (13)$$

where  $DS$  – data structure;  $ds_T$  – data structure type belongs to set of types  $TP$ ;  $DS_F$  – set of data structure fields  $fld^i$ .

$$fld^i = \langle fld_N, fld_T, fld_V \rangle, fld_T \in TP, \quad (14)$$

where  $fld_N$  – field name;  $fld_T$  – field type belongs to set of types  $TP$ ,  $fld_V$  – field value of type  $fld_T$ .

$$(\forall tp \in TP)tp = TP_A | TP_C = \{fld_T\} | TP_L = [TP] \quad (15)$$

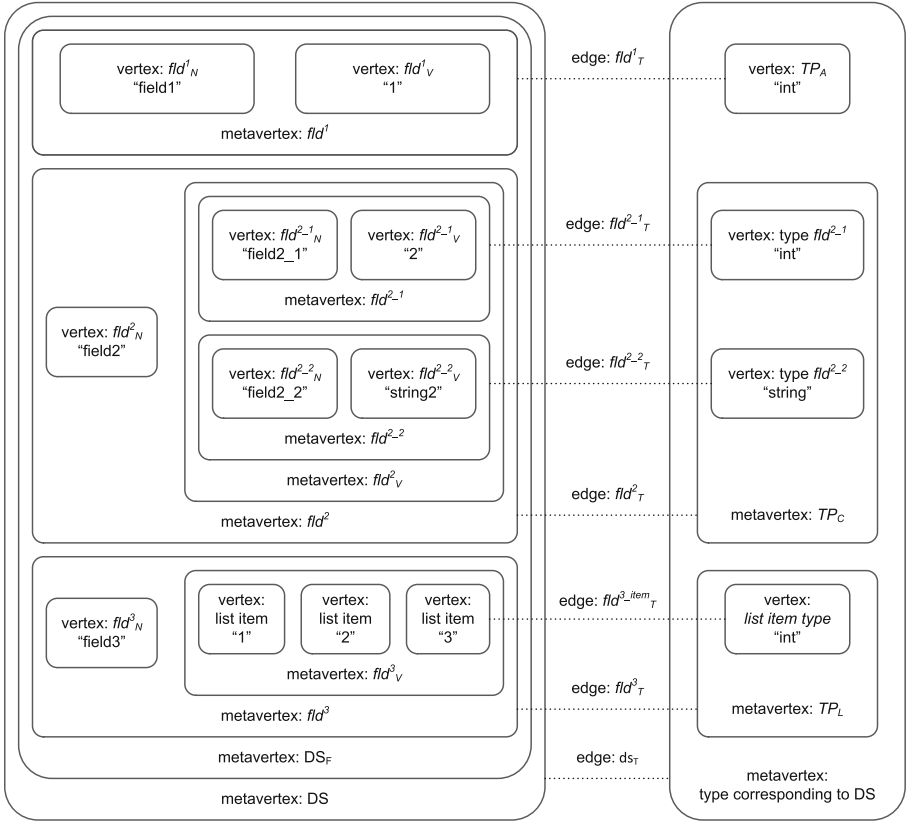
Every type  $tp$  belongs to set of types  $TP$  must be either atomic type  $TP_A$  or complex type  $TP_C$  or list (collection) type  $TP_L$ . The atomic type  $TP_A$  corresponds to the only value. The complex type  $TP_C$  contains set of corresponding field types  $fld_T$ . The list type  $TP_L$  is a collection of elements of any type.

The example showing one of the possible cases of metagraph representation of object-oriented data structure is given in Fig. 3. This example is structured in such a way to cover all possible cases represented by formulas (13–15).

Data structure  $DS$  and its corresponding type are represented as a metaverices bound with edge  $ds_T$ . The set of data structure fields  $DS_F$  (also represented as a metavertex) consists of three fields  $fld^1$ ,  $fld^2$  and  $fld^3$ .

Field  $fld^1$  with name “field1” corresponds to the atomic type “int” with value “1”. Field  $fld^1$  is represented as a metavertex, field name  $fld_N^1$  and value  $fld_V^1$  are represented as inner vertices. The field type is represented as edge  $fld_T^1$  bound field metavertex with atomic type  $TP_A$  vertex.

Field  $fld^2$  with name “field2” corresponds to the complex type consists of fields “field2\_1” of type “int” with value “2” and “field2\_2” of type “string” with value “string2”. Field  $fld^2$  is represented as a metavertex, field name  $fld_N^2$  is represented as inner vertex and value  $fld_V^2$  is represented as inner metavertex containing metaverices  $fld^{2-1}$  and  $fld^{2-2}$  corresponding to subfields “field2\_1” and “field2\_2” with their values. Field  $fld^2$  type is represented as edge  $fld_T^2$  bound field metavertex with complex type  $TP_C$  metavertex. The  $TP_C$  metavertex contains inner vertices corresponding to subfields



**Fig. 3.** Metagraph representation of object-oriented data structure

$fld^{2-1}$  and  $fld^{2-2}$  types. The edges  $fld_T^{2-1}$  and  $fld_T^{2-2}$  bound subfields  $fld^{2-1}$  and  $fld^{2-2}$  metavertrices with corresponding subtypes vertices.

Field  $fld^3$  with name “field3” corresponds to the list (collection) type “list of int” with value “1, 2, 3”. Field  $fld^3$  is represented as a metavertex, field name  $fld_N^3$  is represented as inner vertex and value  $fld_V^3$  is represented as inner metavertex corresponding to the list containing vertices corresponding to the list items. The field type is represented as edge  $fld_T^3$  bound field metavertex with list (collection) type  $TP_L$  metavertex. The  $TP_L$  metavertex contains inner vertex corresponds to the list item type. List items bound with list item type with  $fld_T^{3-item}$  edge (shown only for list item “3” in order not to clutter the figure).

According to formula (9), the metagraph fragment is a set of elements each of which is vertex, edge or metavertex. Thus, Fig. 3 represents metagraph fragment.

Given example shows that object-oriented data structure may be represented using metagraph approach without losing detailed information. Thus metagraph agent transformation mechanism is available for metagraph representation of object-oriented

data. And this makes possible to use object-oriented data as a measure in the hybrid multidimensional-ontological data model.

## 8 The Hybrid Multidimensional-Ontological Data Model

Having considered the metagraph approach we can now propose the data model meeting the “hybrid multidimensional-ontological data model requirements”.

To meet the requirement 1, we can now define the lightweight ontology in form of metagraph. According to the definition the lightweight ontology corresponds to the annotated flat graph that can be easily represented as a special case of metagraph. The concepts of ontology correspond to the metagraph vertices. Semantic relation corresponds to the metagraph edges. According to metagraph model definition, it is possible to annotate concepts vertices and relations edges with any required attributes, in particular, to annotate relations edges with relation type attribute:  $relation\_type = \{is\_a, part\_of, \dots\}$ . The whole lightweight ontology corresponds to metavertex containing sets of concepts vertices and relations edges. According to formula (3), hypercube dimension may be represented in the form of lightweight ontology. Thus, in terms of the metagraph approach according to formulas (1, 3, 6 and 10):

$$hcd_i \equiv OL = \langle T, \mathfrak{R} \rangle, hcd_i \equiv OL \equiv mv_i, T \equiv V, \mathfrak{R} \equiv E, \quad (16)$$

where  $hcd_i$  – hypercube dimension;  $OL$  – lightweight ontology;  $T$  – set of lightweight ontology concepts;  $\mathfrak{R}$  – set of lightweight ontology semantic relations;  $mv_i$  – metagraph metavertex;  $V$  – set of annotated metagraph vertices;  $E$  – set of annotated metagraph edges.

To meet the requirement 2, we can define measure as a metagraph fragment. Numerical and textual information may be represented in form of metagraph vertex which is kind of metagraph fragment. The representation of object-oriented data structures in form of metagraph fragment is discussed in the previous section.

According to formula (16) reference to the dimension element corresponds to metagraph vertex because  $T \equiv V$ . Thus, in terms of the metagraph approach according to formulas (4 and 9):

$$hcf_j = \left\langle \left\{ hcd_i^{ref} \right\}, \left\{ msr_n \right\} \right\rangle, hcd_i^{ref} \equiv v_k, msr_n \equiv MG_i, \quad (17)$$

where  $hcf_j$  – hypercube fact;  $hcd_i^{ref}$  – reference to the dimension element;  $msr_n$  – measure;  $v_k$  – metagraph vertex;  $MG_i$  – metagraph fragment.

The hypercube aggregation rule in the hybrid model corresponds to formula (5). But instead of aggregation function, the metagraph agent is used for aggregation:

$$hcr_k : \{hcf_{OUT}\} = ag^{MG}(\{hcf_{IN}\}, HCD^{ag}), HCD^{ag} \subset HCD, \quad (18)$$

where  $hcf_{OUT}$  – output (aggregated) facts;  $ag^{MG}$  – metagraph agent used for aggregation;  $hcf_{IN}$  – input (non-aggregated) facts;  $HCD^{ag}$  – subset of hypercube dimensions used in aggregation.

If metagraph agent is used in form of function agent according to formula (11) then the combination of input (non-aggregated) facts (in form of metagraph fragment according to formula (17)) and hypercube dimensions (in form of metagraph metavertex according to formula (16)) corresponds to the input parameter metagraph  $MG_{IN}$ . The output (aggregated) facts in form of metagraph fragment are generated according to a function defined by abstract syntax tree  $AST$ .

If metagraph agent is used in form of rule agent according to formula (12) then the combination of input (non-aggregated) facts (in form of metagraph fragment according to formula (17)) and hypercube dimensions (in form of metagraph metavertex according to formula (16)) corresponds to the working metagraph parameter  $MG$ . Open rules are used for generating output (aggregated) facts in form of metagraph fragment.

In case of rule agent, the closed rules may be used for updating dimensions, facts, and measures which are an extension of the traditional multidimensional model.

Summing up it can be noted that formalization of traditional ontological and multidimensional models with formulas (1–5) also valid for the proposed hybrid model. But the formulas (16–18) represent the detailed elements of the multidimensional model (using the ontological model as supporting) in terms of metagraph approach. Thus, the proposed hybrid model makes numerical, textual and object-oriented information possible for using as multidimensional model measures.

## 9 Predicate Representation of Metagraph Model

In previous sections, the formal definition and graphical examples of the metagraph model were defined. But to successfully operate with the metagraph model we also need textual representation. As such representation, we use predicate model close to logical programming languages e.g. Prolog.

The classical Prolog uses following form of predicate:

$$predicate(atom_1, atom_2, \dots, atom_N)$$

We used an extended form of predicate where along with atoms predicate can also include key-value pairs and nested predicates:

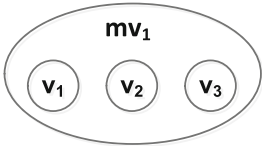
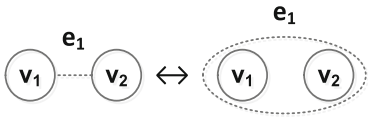
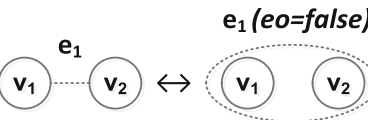

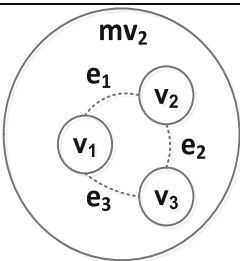
$$predicate(atom, \dots, key = value, \dots, predicate(\dots), \dots)$$

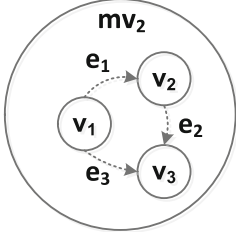
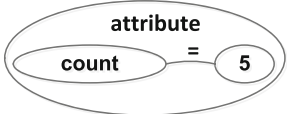
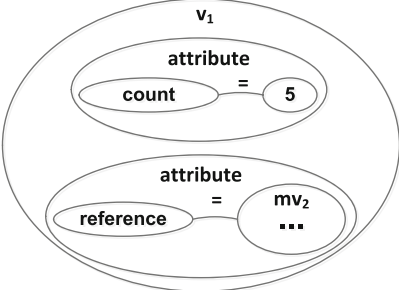
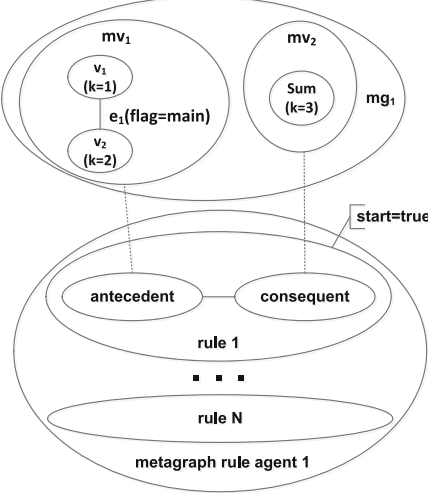
The mapping of metagraph model fragments into predicate representation is shown in Table 1.

Case 1 shows the example of metavertex  $mv_1$  which contains three nested disjoint vertices  $v_1, v_2,$  and  $v_3$ . The predicate corresponds to metavertex, nested vertices are isomorphic to atoms that are parameters of the predicate. As the name of the predicate, “Metavertex” is used as the corresponding element of the metagraph model. Key-value parameter “Name” is used to set the name of metavertex. This case is simplest since nested vertices are disjoint.

Case 2 shows metagraph edge which may be represented as a special case of metavertex containing source and destination vertices. The metagraph edge is represented as a predicate with the name “Edge”. The source and destination vertices are represented as predicate atom parameters.

**Table 1.** Predicate representation of metagraph model

Case №	Metagraph representation	Predicate representation
1		Metavertex(Name= $mv_1, v_1, v_2, v_3$ )
2		Edge(Name= $e_1, v_1, v_2$ )
3		Edge(Name= $e_1, v_1, v_2, eo=false$ )
4		4.1. Edge(Name= $e_1, v_1, v_2, eo=true$ ) 4.2. Edge(Name= $e_1, v_s=v_1, v_e=v_2, eo=true$ )
5		Metavertex(Name= $mv_2, v_1, v_2, v_3,$ Edge (Name= $e_1, v_1, v_2$ ), Edge(Name= $e_2, v_2, v_3$ ), Edge(Name= $e_3, v_1, v_3$ ))

<p>6</p>		<p>Metavertex(Name=mv<sub>2</sub>, v<sub>1</sub>, v<sub>2</sub>, v<sub>3</sub>, Edge(Name=e<sub>1</sub>, v<sub>S</sub>=v<sub>1</sub>, v<sub>E</sub>=v<sub>2</sub>, eo=true), Edge(Name=e<sub>2</sub>, v<sub>S</sub>=v<sub>2</sub>, v<sub>E</sub>=v<sub>3</sub>, eo=true), Edge(Name=e<sub>3</sub>, v<sub>S</sub>=v<sub>1</sub>, v<sub>E</sub>=v<sub>3</sub>, eo=true))</p>
<p>7</p>		<p>Attribute(count, 5)</p>
<p>8</p>		<p>Vertex(Name=v<sub>1</sub>, Attribute(count, 5), Attribute(reference, mv<sub>2</sub>) ...)</p>
<p>9</p>		<p>RuleAgent (Name='metagraph rule agent 1', WorkMetagraph=mg<sub>1</sub>, Rules( Rule(Name='rule 1', start=true, Condition(WorkMetagraph=mv<sub>1</sub>, Vertex(Name=v<sub>1</sub>, Attribute(k, \$k1)), Vertex(Name=v<sub>2</sub>, Attribute(k, \$k2)), Edge(v<sub>1</sub>, v<sub>2</sub>, Attribute(flag, main))) Action(WorkMetagraph=mv<sub>2</sub>, Add(Vertex(Name=Sum, Attribute(k, Eval(\$k1+\$k2))))))</p>

Case 3 also shows metagraph edge which fully complies with the formal definition of an undirected edge according to formula (8) including direction flag parameter (eo = false).



Case 4 shows an example of directed edge. Direction flag parameter is also used. The source and destination vertices may be represented as predicate atom parameters (case 4.1) or as predicate key-value parameters (case 4.2).

Case 5 shows an example of metavertex  $mv_1$  which contains three nested vertices  $v_1$ ,  $v_2$  and  $v_3$  joined with undirected edges  $e_1$ ,  $e_2$ , and  $e_3$ . Edges are represented with separate predicates that are nested to the metavertex predicate. Case 6 is similar to case 5 unless edges  $e_1$ ,  $e_2$ , and  $e_3$  are directed.

The attribute may be represented as a special case of metavertex containing name and value. Case 7 shows simple numeric attribute representation. Case 8 shows an example of vertex  $v_1$  containing numeric attribute and reference attribute that refers to the metavertex  $mv_2$ . The attribute is represented as a predicate with the name "Attribute".

Case 9 shows an example of metagraph rule agent "metagraph rule agent 1" representation (the predicate with the name "RuleAgent" is used). As a work metagraph,  $mg_1$  is used (parameter "WorkMetagraph"). The "Rules" predicate contains rules definition (nested predicate "Rule" is used). As a start rule "rule 1" is used which is defined by "start = true" parameter. Predicate "Condition" corresponds to the rule condition. Parameter "WorkMetagraph" of predicate "Condition" contains a reference to the tested metavertex  $mv_1$ . The condition tests that metavertex  $mv_1$  contains vertices  $v_1$  and  $v_2$  with attribute  $k$ . Founded values of  $k$  attribute of vertices  $v_1$  and  $v_2$  are assigned to the  $\$k1$  and  $\$k2$  variables. Vertices  $v_1$  and  $v_2$  should be joined with edge containing attribute "flag = main". If the condition is true and the metagraph fragment is found then the action is performed (action is defined by predicate "Action"). Parameter "WorkMetagraph" of predicate "Action" contains a reference to the result metavertex  $mv_2$ . The example action contains adding the new element (that is defined by predicate "Add"). The vertex "Sum" is added containing attribute " $k = \$k1 + \$k2$ ". Predicate "Eval" is used to define the calculated expression.

Thus, we have defined a predicate description of all the main elements of the metagraph data model.

It should be noted that proposed predicate model is homoiconic. Since predicate approach is used as for metagraph data model definition and for metagraph agents definition then high-level metagraph agents may change the structure of low-level metagraph agents by modifying their predicate definition.

Proposed predicate model may be considered as a physical data model for metagraph approach implementation.

## 10 Conclusions

Today multidimensional model is used in a great number of information systems. The advantages of the multidimensional model are worldwide recognized. But the traditional multidimensional model is oriented for numerical measures usage.

The ontological model may be considered as a convenient supporting model for describing parts of multidimensional model e.g. dimensions.

Metagraph approach allows describing complex graph models. Using metagraph agents it is possible either to generate the output metagraph based on the input metagraph (using open rules) or to modify the metagraph (using closed rules).

Using metagraph approach it is possible to describe different kinds of information such as object-oriented information in form of complex graph.

The hybrid multidimensional-ontological data model is proposed on the base of metagraph approach. Using this approach, it is possible to use numerical, textual and object-oriented information as multidimensional model measures. The aggregation of such measures is performed using metagraph agents.

## References

1. Blondé, W., Antezana, E., Mironov, V., Schulz, S., Kuiper, M., De Baets, B.: Using the relation ontology Metarel for modelling Linked Data as multi-digraphs. *Semant. Web* **5**(2), 115–126 (2014)
2. Qu, Q., Qiu, J., Sun, C., Wang, Y.: Graph-based knowledge representation model and pattern retrieval. In: FSKD, vol. 5, pp. 541–545 (2008)
3. Gruber, T.: A translation approach to portable ontology specifications. *Knowl. Acquisition* **5**(2), 199–220 (1993)
4. Gavrilova, T., Chernigovskaya, T.: Cognitive aspects of visual knowledge base design. In: Proceedings of the International Conference PEG. Intelligent Computer and Communications Technology (Teaching & Learning for the 21-st Century), Great Britain, pp. 174–181 (1999)
5. Allemang, D., Hendler, J.: *Semantic Web for the Working Ontologist: Effective Modelling in RDFS and OWL*, 2nd edn. Elsevier, New York (2011)
6. Defining N-ary Relations on the Semantic Web. W3C Working Group Note, 12 April 2006. <http://www.w3.org/TR/swbp-n-aryRelations>
7. Codd, E., Codd, S., Salley, C.: Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. [http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem\\_dwh/lit/Cod93.pdf](http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf)
8. Mansmann, S., Scholl, M.: Extending the multidimensional data model to handle complex data. *J. Comput. Sci. Eng.* **2**(1), 125–160 (2007)
9. Semantic OLAP: Semantic Mediawiki Extension. [https://www.mediawiki.org/wiki/Extension:Semantic\\_OLAP](https://www.mediawiki.org/wiki/Extension:Semantic_OLAP)
10. Basu, A., Blanning, R.: *Metagraphs and Their Applications*. Springer, Heidelberg (2007). <https://doi.org/10.1007/978-0-387-37234-1>
11. Samohvalov, E., Revunkov, G., Gapanyuk, Y.: Metagraphs for information systems semantics and pragmatics definition. In: Herald of Bauman Moscow State Technical University, vol. 1, no. 100, pp. 83–99 (2015)

# PosDB: A Distributed Column-Store Engine

George Chernishev<sup>1,2(✉)</sup>, Viacheslav Galaktionov<sup>1</sup>, Valentin Grigorev<sup>1</sup>,  
Evgeniy Klyuchikov<sup>1</sup>, and Kirill Smirnov<sup>1</sup>

<sup>1</sup> Saint-Petersburg University, Saint-Petersburg, Russia  
chernishev@gmail.com, viacheslav.galaktionov@gmail.com,  
valentin.d.grigorev@gmail.com, evgeniy.klyuchikov@gmail.com,  
kirill.k.smirnov@math.spbu.ru

<sup>2</sup> JetBrains Research, Saint-Petersburg, Russia  
<http://www.math.spbu.ru/user/chernishev/>

**Abstract.** In this paper we present a novel disk-based distributed column-store, describe its architecture and discuss a number of technical solutions. Our system is essentially a query engine which was written completely from scratch. It is aimed for shared-nothing environments and supports different forms of parallel query processing.

Query processing in PosDB is organized according to the classic Volcano pull-based model which is adapted for the column-store case. Currently, we support late materialization only, and therefore employ a join index data structure to represent positional information. In our system query plan can consist of both positional and value operators. PosDB has about a dozen of core operators among which several variants of selections and joins, aggregation. We also have several operators that ensure intra-query parallelism and operators for network interoperability. In its current state the system is fully capable of processing the Star Schema Benchmark in a local and distributed environment.

## 1 Introduction

A column-store DBMS is a system which stores each attribute in a separate file. Approximately ten years ago, this approach experienced a sharp rise in popularity in both academic and industrial communities [3, 4, 6]. There were multiple reasons for the newly found interest in column-stores. Firstly, column-stores were able to excel in query processing in OLAP environments, which became popular several years earlier. Next, column-stores were able to utilize the changes that had accumulated in hardware design over the years. Consequently, these systems were able to use contemporary hardware more efficiently than classic row-stores. This interest led to a surge of studies and produced a number of research prototypes. In turn, this resulted in creation of a number of commercial systems. Currently, a lot of major DBMS vendors offer their own column-oriented products. However, column-stores still continue to be an active research field.

During the boom of the 2000s various studies involved different aspects of column-oriented DBMSes: column-specific operators, query processing schemes,

column-specific (adaptive) indexing, compression, data reorganization, early and late materialization, hardware efficiency and many more. However, some topics received less attention, among them were query optimization, physical design, and adaptivity in a broader sense — adaptivity of plans, data and operators. Moreover, the issues related to distributed column-stores were left completely untouched.

The reason for the lack of studies is the absence of a research prototype with distributed capabilities. Both prominent earlier prototypes — C-Store [13] and MonetDB [11] — are centralized, as are the majority of contemporary research systems. There are two exceptions — ScaMMDB [15] and DCODE [12], which are essentially attempts to distribute MonetDB by implementing a network layer on top of it. However, these systems have two significant drawbacks. Firstly, the underlying DBMS is an in-memory system. Despite the rise of interest in the in-memory solutions, disk-based systems are still relevant. Thus, to the best of our knowledge there are no distributed disk-based column-stores which would allow to conduct research related to distributed query processing. Secondly, the distribution obtained in such a way cannot be considered a “true” distribution, because it may be restricted by the architecture of the existing DBMS. For example, the set of admissible query plans may exclude good ones due to architectural restrictions.

In our previous papers [7,8], we described possible benefits of a distributed column-store and sketched its design. In this paper, we continue our work and describe the first version of our system. It is essentially a column-store query engine, designed for research purposes such as the study of distributed query processing and adaptivity of data, operators, query plans.

## 2 Column-Store Basics

The core idea of the column-store approach is to store the relations in columnar form. This means that each attribute is stored apart from all other attributes of the same table and kept in a separate file. The data is not only stored but also processed (at least partially) in this form.

This differs greatly from the classic DBMS mode of operation and requires a different approach. However, row-stores have been around for more than 40 years, and over this time, a lot of experience has been accumulated in the form of approaches, technical solutions, and principles. It is highly desirable to make use of this experience when designing column-store systems. Nevertheless, a number of column-stores properties should be taken into account:

- Column-store operators usually work with single attributes instead of tables. Moreover, they can exchange not only values but also ordinal positions of these values in the corresponding tables.
- Consequently, there are not only value-oriented operators but also position-oriented or mixed ones. For example, consider a query that extracts all IDs of rows which satisfy two predicates on two different attributes. A possible

query plan is to filter both attributes, obtain lists of positions, and invoke the positional AND operator.

- Some of the operators may have to access the corresponding attribute values if given position lists as input.
- Tuple reconstruction has to be performed at some point of the query plan. This reconstruction is needed to prepare a tuple for returning to the user. This preparation is essentially transformation of data from internal representation. Note that unlike row-stores, this step is mandatory in column-stores.
- Another problem is the so-called materialization. There are two basic options: early and late materialization. The former is similar to row-stores, where the value is used from the start of processing. The latter implies that a position would be substituted with a value at a later point in time. The choice of the point of materialization is an optimization problem.

These differences define the architecture of a column-store DBMS. Back in the early 2000s there were several column-store research projects which successfully designed such systems. We can summarize their findings as follows:

- A new algebra of operations and novel cost models.
- Materialization strategy selection problem: two primary approaches and many more advanced techniques.
- More complex query plans — in some architectures it is necessary to represent query plan as a directed acyclic graph instead of a tree.
- Novel approaches to operator design. New operators appear and new implementations for the old ones are possible.
- Compression is ubiquitous in this class of systems — data is not only stored, but also processed in compressed form.

See studies [3, 4, 6] for more detailed introduction into column-store systems.

### 3 Motivation and Aims

To the best of our knowledge, there is no distributed disk-based column-store suitable for the research purposes. Our goal is to develop such a system and use it to study advanced query processing and adaptivity in a distributed environment.

### 4 Existing Column-Store Systems

There are several major research prototypes and a large number of commercial implementations [3, 6, 11]. Let us describe the research ones briefly [8]:

- C-Store is a disk-oriented column-store database. It supports late materialization, special join operators, different compression methods, as well as operating on compressed data.

- MonetDB is an in-memory column-store database. Its goal is efficient hardware usage: minimization of CPU cache misses and exploitation of hardware parallelism, e.g. SIMD instructions. MonetDB features a special algebra operating on BATs (columns), and operators designed for efficient hardware usage. Another interesting result is the adaptive indexing techniques, where index is an additional result of query execution.
- Supersonic [2] is an open-source columnar query engine which is oriented towards efficient data processing. It is an in-memory query engine which focuses on cache consciousness, vectorized execution, instruction pipelining, and the usage of SIMD instructions.
- Peloton [5] is a new open-source in-memory DBMS. It is aimed for efficient hybrid transaction-analytical processing.

All of these are centralized database column-stores. However, there are both commercial and academic distributed systems based on some of them. The latter [12, 15] were discussed in the introduction.

It is important to mention that there are a lot of open-source NoSQL systems which are considered column-stores: HBase, Cassandra, Druid and many others. However, non-relational systems are a very different field of study, they rely on different principles and use different mechanisms for data processing, so they do not suit our purposes.

## 5 Architecture

We have developed a disk-oriented column-store engine aimed for shared-nothing environments. In our system query execution is based on the pull-based Volcano model [10] with block-oriented processing. This means that a query plan is represented by a tree of operators where each operator supports the following interface:

```

open(): initialize operator processing,
next(): the processing itself — construct the next result block, possibly request-
       ing the data (values or positions) from children nodes,
close(): terminate operator processing, free resources.

```

We have also added a special `rewind()` method to restart an operator without full resource deallocation.

Right now, our system supports only the late materialization strategy. A join index data structure [14] is used to describe the correspondence between records of different tables which arises after successful join.

PosDB is able to operate in both centralized and distributed modes. In the distributed mode all nodes behave in the same way. Each host can process a query plan supplied either by user or by another host. As of now, only the part before materialization can be transferred. That includes joins, cross-products, filtering, and positional operators.

A special attention has been given to handling network-related errors. If a server node disconnects in the middle of interaction, the client node tries

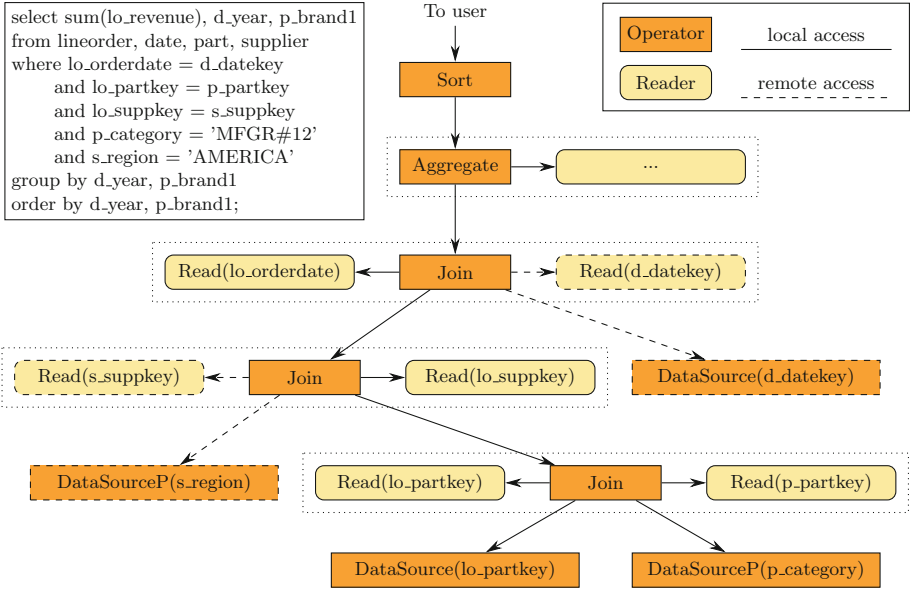


Fig. 1. Example plan: Star schema benchmark, Query 2

to reconnect with a certain delay until it succeeds. Once the connection has been reestablished, the interaction will proceed from where it was interrupted, if possible. If not, the query plan will be sent again and a required amount of data will be skipped.

To illustrate PosDB architecture we present the plan of query 2 from the Star Schema Benchmark in Fig. 1. This query plan consists of operators and readers. A reader is a special entity which is always coupled with some operator and is used to acquire data. The plans of our system need readers due to the reliance on the late materialization approach. Here, you can also see relational operators, a sorting operator, and a data source operator. A data source operator is a leaf operator which performs data filtering and returns a list of positions.

## 6 Implementation Details

PosDB is implemented in the C++ programming language and adheres to the C++11 standard. We have a set of tests (about 140) built using the Google Test framework. Currently, our system is quite light-weight — there is about 900 KB of our source code.

## 7 Current State: Present and Missing Features

**Present Features.** Let us describe the current features of our system:

- Our system is distributed; with all nodes equally capable to process any allowable query. Currently, the distribution allows to perform any part of query plan before the materialization point on some other host.
- There is plan-level parallelism in our system: we have implemented an n-ary node which processes its subtrees in separate threads and merges their results into one stream in arbitrary order.
- Our system supports late materialization, which allows our plans to operate on positions until values are requested by operators or tuple reconstruction.
- Our system possesses a broad range of operators, both value-oriented and position-oriented. Among value-oriented ones are three classic implementations of equi-join operator (hash, sort-merge, nested loop), several variants of aggregation, and cross-product. We implemented all of the operators which were required to run the Star Schema Benchmark [1].
- PosDB in its current state is capable of processing SSB in either a centralized or distributed manner. Results of preliminary performance evaluation for centralized and distributed environments are reported in paper [9].

**Missing Features.** We do not support data-modifying queries and, consequently, transactions. SSB also does not support them because it is aimed for OLAP applications. Since this is the initial version of our system, some features are missing:

- subqueries and other more advanced parts of SQL,
- a parser and an optimizer,
- a buffer manager,
- capability to process or operate directly on compressed data,
- any vectorized or column-specific operators, e.g. invisible join,
- early materialization.

Since our system cannot take text queries as input, query plans have to be specified in the source code. Not supporting early materialization is a considerable drawback for a column-store, but late materialization alone already allows (limited) experimentation with distributed query plans.

The first goal of our project was to implement the backbone for processing of all SSB queries — SPJs with aggregation and sorting, and we have achieved it. We will continue with our work and implement some of the missing features, with our primary objective being the improvement of distributed processing.

## 8 Conclusion

In this paper we have presented our column-oriented query engine for experimental study of query processing and adaptivity in a distributed environment. First, we described a niche for such a system and justified its relevance and



usefulness for the community. Then we outlined its architecture, features, and described some of the technical solutions used.

Currently, our system is capable of processing the Star Schema Benchmark, a standard benchmark for evaluation of column-stores, both locally and in a distributed fashion.

## References

1. O'Neil, P.E., O'Neil, E.J., Chen, X.: The Star Schema Benchmark (SSB) (2009). <http://www.cs.umb.edu/~poneil/StarSchemaB.PDF>. Accessed 20 July 2012
2. Google Supersonic Library (2017). <https://code.google.com/archive/p/supersonic/>. Accessed 12 February 2017
3. Abadi, D., Boncz, P., Harizopoulos, S.: The Design and Implementation of Modern Column-Oriented Database Systems. Now Publishers Inc., Hanover, massachusetts (2013)
4. Abadi, D.J., Boncz, P.A., Harizopoulos, S.: Column-oriented database systems. *Proc. VLDB Endow.* **2**(2), 1664–1665 (2009)
5. Arulraj, J., Pavlo, A., Menon, P.: Bridging the Archipelago between row-stores and column-stores for hybrid workloads. In: Proceedings of the 2016 International Conference on Management of Data, SIGMOD 2016, pp. 583–598 (2016)
6. Chernishev, G.: Physical design approaches for column-stores. *SPIIRAS Proceedings* **30**, 204–222 (2013)
7. Chernishev, G.: Towards self-management in a distributed column-store system. In: Morzy, T., Valduriez, P., Bellatreche, L. (eds.) *ADBIS 2015*. CCIS, vol. 539, pp. 97–107. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-23201-0\\_12](https://doi.org/10.1007/978-3-319-23201-0_12)
8. Chernishev, G.: The design of an adaptive column-store system. *J. Big Data* **4**(1), 21 (2017)
9. Chernishev, G., Galaktionov, V., Grigorev, V., Klyuchikov, E., Smirnov, K.: A study of PosDB performance in a distributed environment. In: Proceedings of the 2017 Software Engineering and Information Management, SEIM 2017 (2017)
10. Graefe, G.: Query evaluation techniques for large databases. *ACM Comput. Surv.* **25**(2), 73–169 (1993)
11. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K.S., Kersten, M.L.: MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.* **35**(1), 40–45 (2012)
12. Liu, Y., et al.: DCODE: A distributed column-oriented database engine for big data analytics. In: Khalil, I., Neuhold, E., Tjoa, A.M., Da Xu, L., You, I. (eds.) *CONFENIS/ICT-EurAsia -2015*. LNCS, vol. 9357, pp. 289–299. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-24315-3\\_30](https://doi.org/10.1007/978-3-319-24315-3_30)
13. Stonebraker, M., et al.: C-Store: A column-oriented DBMS. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 553–564. VLDB Endowment (2005)
14. Valduriez, P.: Join indices. *ACM Trans. Database Syst.* **12**(2), 218–246 (1987)
15. Zhang, Y., Xiao, Y., Wang, Z., Ji, X., Huang, Y., Wang, S.: ScaMMDB: Facing challenge of mass data processing with MMDB. In: Chen, L., et al. (eds.) *APWeb/WAIM -2009*. LNCS, vol. 5731, pp. 1–12. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03996-6\\_1](https://doi.org/10.1007/978-3-642-03996-6_1)

# Microservices: How To Make Your Application Scale

Nicola Dragoni<sup>1,5</sup>, Ivan Lanese<sup>2</sup>, Stephan Thordal Larsen<sup>1</sup>,  
Manuel Mazzara<sup>3</sup>(✉), Ruslan Mustafin<sup>3</sup>, and Larisa Safina<sup>3,4</sup>

<sup>1</sup> Technical University of Denmark, Kongens Lyngby, Denmark  
ndra@dtu.dk, stephan@thordal.io

<sup>2</sup> University of Bologna/INRIA, Bologna, Italy  
ivan.lanese@gmail.com

<sup>3</sup> Innopolis University, Innopolis, Russian Federation  
{m.mazzara,r.mustafin,l.safina}@innopolis.ru

<sup>4</sup> University of Southern Denmark, Odense, Denmark

<sup>5</sup> Örebro University, Örebro, Sweden

**Abstract.** The microservice architecture is a style inspired by service-oriented computing that has recently started gaining popularity and that promises to change the way in which software is perceived, conceived and designed. In this paper, we describe the main features of microservices and highlight how these features improve scalability.

## 1 Introduction

History of programming languages and paradigms has been characterized in the last few decades by a progressive shift towards distribution, modularization and loose coupling, with the purpose of increasing code reuse and robustness [5]. This necessity has been dictated by the need of increasing software quality, not only in safety and financial-critical applications, but also in more common off-the-shelf software packages.

Service oriented architectures (SOAs) can be seen as a step in this direction, where the need for code reuse and robustness was coupled with the need for interoperability between heterogeneous information systems, possibly belonging to different companies. This brought up the idea of a *service* as a software entity interacting with other software entities via message passing communications using standard data formats and protocols (e.g., XML, SOAP and HTTP) and well-defined interfaces.

Microservices are a further step along this road, emphasizing the use of small services, called indeed microservices, and moving the service oriented techniques from system integration to system design, development and deployment.

The microservice architecture [9] is built on a few basic principles:

- *Bounded Context*. First introduced in [11], bounded context means that related functionalities are combined into a single business capability, and each microservice implements one such capability. In this way there is a perfect

alignment between business capabilities and system structure, making it easy, e.g., to know where a functionality is, in order to update or fix it.

- *Size*. The focus on small size is a crucial novelty of microservices w.r.t. the previous SOAs. Idiomatic use of microservice architectures suggests that if a service is too large, it should be refined into two or more services, thus preserving granularity and maintaining focus on providing a single business capability only. The small size brings major benefits in terms of service maintainability and extendability: a small service can be easily modified, and if needed rebuilt from scratch with limited resources and in limited time.
- *Independency*. This concept encourages loose coupling by stating that each microservice in microservice architectures is operationally independent from others, and the only form of communication between services is through their published interfaces. This is fundamental since this allows one to change, fix or upgrade a microservice without compromising the system correctness, provided that the interfaces are preserved. High cohesion is also encouraged, since related functionalities should be provided by a unique microservice, so that any update related to those functionalities only affects the corresponding microservice.

The shift towards microservices is a sensitive matter these days. Several companies are involved in a major refactoring of their back-end systems to accommodate the advantages of the new paradigm [7]. Other companies just start their business model developing software following the microservice paradigm since day one. We are in the middle of a major change in the view in which software is intended, and in the way in which capabilities are organized into components, and industrial systems are conceived. In the next section we highlight another advantage of microservices: scalability, for performance, fault tolerance or availability reasons.

## 2 Scalability

Scalability is one of the key features provided by the microservice paradigm. In this section, we aim at giving an overview on how microservice characteristics naturally contribute to system scalability. We emphasize that while frequently scalability is needed for performance reasons, to cope with high load, scalability can also be used to ensure availability and fault tolerance. According to the reason why scalability is needed, slightly different approaches need to be used, as we will emphasize below.

**Distribution.** Distribution is not an original feature of microservices, since, e.g., SOAs are distributed as well. However, thanks to their small size, microservices take this characteristics to an extreme: each business capability, including their functionalities and the related data, is realized by an independent service, which can be deployed on a host possibly different from the one of other microservices of the same application. As first result, this causes a natural distribution of the

workload that can make the system significantly more efficient than a monolith [2]. Distribution also makes microservice architectures highly available, since the failure of a single microservice does not necessarily result in the failure of other microservices. Distribution can also utilize locality and locate services closer to the clients they serve, resulting in better geographical scalability [2, 18].

**Non-uniform Scaling.** Typically, when monolithic architectures are exposed to growing load, it is difficult to locate which components of the system are actually affected, since the system runs within a single process. This means that, although only a single component may be experiencing load, the whole monolith will need to scale, e.g. by replication or vertical scaling. Even if it is known which is the component that is experiencing load, it is difficult to scale it in isolation. The same reasoning may apply to SOAs: services in SOAs may be large, frequently hiding a whole monolithic application behind a service-oriented interface, hence they may only scale at a large granularity. The same applies when scalability is needed to implement high availability: if only some components of a monolith or of a large service are required to be highly available, the whole monolith/large service will have to be highly available.

Since microservices are implemented and deployed independently of each other, i.e. they run within independent processes, they can be monitored and scaled independently, as shown by the example below.

*Example.* A simplified illustration showcasing the benefits of scaling a microservice architecture, compared to a monolithic architecture, is given in Fig. 1. Both the systems implement componentization of software, the monolith utilizing regular software components, such as libraries, and the microservice architecture utilizing microservices, i.e. *Component x* corresponds to *Service x*. In this scenario *Component/Service 1* is experiencing a load that requires to replicate it to 3 instances. Since the monolith is deployed as a single process, one needs to replicate the whole system, including all 3 components, across three hosts. In a microservice architecture one can simply replicate the single service experiencing load, resulting in allocation of much fewer hosts. The load balancers are in place in both systems to split the load across replicas. However, in the monolith the balancer splits only external requests, while in the case of the microservice architecture it splits both external requests and internal requests between the different microservices, thus allowing for a more uniform load balancing. This happens in particular when external requests may trigger computations which are heavy in a possibly non-uniform way: only balancing external requests may not be enough.

The reliance on *Domain-Driven Design* [11] and the strive towards *high cohesion* means that growing load will typically be delimited to a subset of associated microservices [19]. The specific microservices actually experiencing the growing load can then be scaled, e.g., by relocating them to the more performant hosts or by replicating them across a cluster or on the cloud.

A similar argument applies to the technology adopted for implementing each microservice: the technology used to build a microservice can be chosen in order

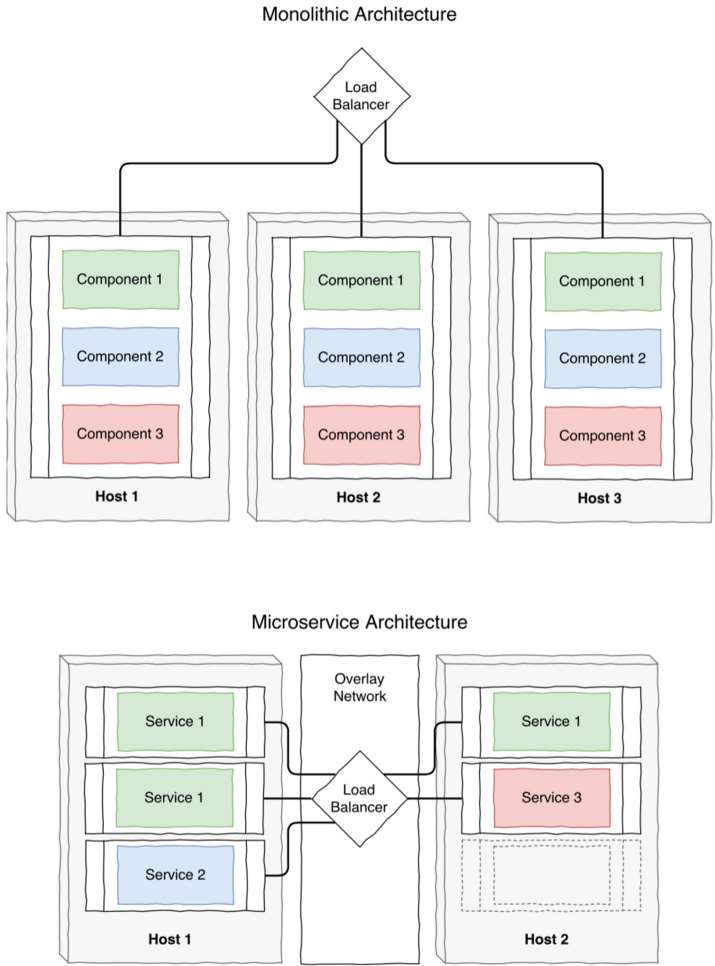


Fig. 1. Scaling in microservices vs monolithic architecture

for it to perform at best. For instance, a computation-intensive microservice might be implemented in C++, while a microservice requiring to handle complex types could be implemented in a language with a sophisticated type system, like Haskell. This is not possible in a monolithic architecture which is typically bound to a single platform and language.

**Portability.** Microservices are typically packaged in *containers*, as provided, e.g., by *Docker* [15] or similar technologies. A container includes the microservice and all its environment (libraries, databases, ...) in a unique entity which can be easily deployed on any platform supporting the chosen container technology, ensuring uniform behavior over heterogeneous platforms (hosts, data-centers and

cloud providers) and isolation w.r.t. other containers (e.g., different microservices can use different versions of the same library without conflicts). The portability ensured by containers enables effortless relocation or replication of a microservice across heterogeneous platforms. Microservice architectures are therefore ideal for scaling a system horizontally, since the microservices can easily be relocated to newly provisioned hosts.

**Elasticity.** The ability to easily replicate individual microservices, coupled with the ability to locate both a single and multiple microservices on a single host, also enables microservice architectures to be elastic, that is to dynamically scale according to the load. Because of this multiple-service-per-host model, deploying a microservice architecture to a dynamically-sized cluster, and in particular on the Cloud, allows it to utilize available resources very efficiently. When the load is high, the system can easily be expanded by exploiting additional hosts dynamically allocated to it in the cluster or new virtual machines on the cloud, and when resources become redundant because of lower load then those hosts can be de-provisioned and removed from the cluster/cloud again. In the same way, the number of service replicas can be increased or shrunk when needed. This feature makes microservices a natural technology for the cloud, and suggests that microservice popularity will continue to grow as far as more and more applications are moved to the cloud.

**Availability.** We have already highlighted some of the ways in which microservices can help availability, but here we summarize the main aspects related to the topic. In general, high availability is achieved by microservices' ability to be replicated and spread across data-centers and geographical distances, allowing them to spread load and cope with failing and congested hardware. Another relevant aspect concerns system update and evolution: while updating a monolithic application requires to stop it and re-deploy it, thus causing a possibly long downtime, replicability and independence allow microservices to solve the problem. First, updating a microservice architecture normally involves just one or a few microservices related to the business capability that needs to be fixed or improved, hence reducing the deployment time. Furthermore, the old and the new version of the same microservice can run side by side, e.g., the old one completing running requests and the new one taking care of new requests. The old one can then be removed when its job is ended. Note that containerization avoids interferences between the two versions of the service, e.g., allowing them to rely on different versions of the same library. This naturally leads to smaller but more frequent updates, in the direction of continuous deployment.

**Robustness.** As for availability, also robustness benefits from using a microservice approach. Indeed, one may replicate microservices as described above to ensure fault tolerance. Fault tolerance however is also naturally improved because of the usage of *containerization* and independent processes. Indeed, a

single microservice is completely isolated from other microservices and can only be affected by them through its defined interfaces or through the resources it relies on. This means that even though some microservices might fail, isolation ensures that other microservices and their environments are not affected. Of course, this requires microservices to implement some fault-tolerant mechanisms that can detect possible failures in microservices they depend on in order to prevent cascading failures.

One should however pay attention that low level interferences may still happen, in particular when multiple microservices are deployed on the same host. Indeed, although their logical environments might be isolated, their physical one is not. If a single microservice consumes all the resources of a host shared with other microservices, those microservices will be affected. Therefore one should be careful when placing microservices together on the same host and take the possible load of each of the microservices into account both before and during operation, ensuring that resources are not exhausted by a single microservice.

**No Silver Bullet.** The description above should clarify how microservices provide a natural way to reach scalability, including availability and fault tolerance. However, this does not come for free: having multiple independent entities introduce some extra administrative overhead, in particular for deployment, administration, monitoring, and security. While there are approaches to mitigate these problems (but still far from satisfactory, at least concerning security), this means that sometimes microservices are not the solution. The description above should help to understand the advantages of microservices, and deciding whether they are a good technique for the problem at hand. We discuss below some concrete cases to further clarify the issue.

### 3 The Language Choice

While microservice architectures can be built using a wide range of technologies, possibly combined into the same system, we think that the use of a dedicated language can simplify the development of microservice systems. Our experience is based on the language Jolie, the only language we are aware of natively supporting microservice architectures. While we refer to [17] for a detailed description of the Jolie language, we recall here its features that come handy for our discussion, and in particular the ones related to the characteristics of microservices above.

In Jolie each program is a microservice, and its description is composed by a behaviour, and some deployment information, concerning how it can communicate with other microservices. In this sense, distribution is inherent in the language, since each microservice makes its functionalities available at a specific URL, and can be invoked by other microservices. Non-uniform scalability can be easily obtained: new microservice instances can be run, and one can easily redirect requests from a single microservice to a load balancer: targets of microservice invocations are a first-class object in Jolie, hence they can be easily and dynamically changed. Primitives for architectural composition such as

redirection or aggregation also help in this direction. The support that Jolie provides to these basic aspects, and the fact that it fully supports the microservice paradigm, ensure that also the other relevant properties hold. Indeed, Jolie has no specific language support for containerization or elasticity, and indeed how such a language support can be provided and whether it would be beneficial for the language or not is an active topic of discussion in the Jolie community. However, Jolie microservices can be easily deployed in Docker [15] containers or on the cloud, hence what said above in this respect holds for Jolie microservices as well. We close this section with a note on robustness: Jolie provides advanced mechanisms for fault notification between different microservices [12]. These mechanisms allow detailed control on whether faults are propagated from one microservice to the ones interacting with it. Indeed, non propagating them allows one to avoid cascading errors, but careful propagation allows one to restore a correct distributed state for the whole system, while preserving the independence of the single microservices. Indeed, each microservice is responsible for restoring its own state, but distributed coordination allows one to ensure global consistency.

## 4 Applications

Microservice architecture has an ideal application where scalability, minimality and cohesiveness are required. Several companies nowadays are moving their monolithic architectures to microservices to reap benefits of scalability. Netflix is one such example - they were one of the pioneers who moved from monolith to microservices [10]. Now Netflix underlying microservice architecture enables them to scale effectively and serve millions of users everyday. Portability was used by Netflix not only to make deployment and relocation easier, but also to automatise the deployment: a deployment tool that knew how to deploy a container, could deploy it no matter what was inside it. Microservice architecture also allowed Netflix to improve robustness and availability by launching a service called Chaos monkey [3] to continuously test for faults within the system. Chaos monkey, as the name suggests, causes chaos inside the system by shutting down various services randomly and observing how the system would adapt to these failures. Despite the fact that Chaos Monkey produces faults on the running system, the system still operates within the limited period of time when engineers are able to respond to the possible crash.

Our research group has investigated another application of the architectural style exploiting the flexibility of the Jolie programming language: the emerging area of smart buildings, with an outlook on IoT and smart cities. Rooms of a building have been equipped with a number of devices and sensors in order to capture the fundamental parameters determining well-being, comfort and livability of humans, such as temperature, humidity, and illumination [22,23]. The purpose is to monitor and optimize working conditions, and the software infrastructure, tightly connected to the hardware, makes use of Jolie and microservices.



The system is designed separating the logic into small components. Each service is responsible for managing one sensor or one specific function. Some services are written in Java for a simpler interaction with devices, and Jolie works as an orchestrator for the entire set of services. There are several advantages in this approach. First of all, *reusability*. The system supports different kinds of sensors, but the central logic of data extraction is unchanged even when sensors are added, removed or substituted. Second, code *readability*, since services are simple components with a simple logic and a clear naming convention. The combination of readability and reusability also leads to *reduced bugs*. *Scalability*, *minimality* and *cohesiveness* are necessary due to the need of connecting sensors and actuators, removing them, adding new ones, managing faults and monitoring the dynamic nature of the infrastructure, especially when mobile devices and “*things*” are part of the system. The *elasticity* of the context has to be managed partially automatically, partially through human intervention from a central control panel, therefore demanding the need for service orchestration and workflow management.

## 5 Microservices and Beyond

The microservice architecture does not build on vacuum and relates to well-established paradigms such as OO and SOA. In [9] a comprehensive survey on recent developments of microservice architecture is presented focusing on the *evolutionary* aspects more than the *revolutionary* ones. The presentation there is intended to help the reader in understanding microservices, their origin and their possible future.

Microservices can be built using a wide range of technologies combined into the same system. However, we support the idea that a language-based approach can simplify development [4]. Jolie is the only language we are aware of that is natively supporting the paradigm. Workflow engines have been around for long [13], and workflow languages capable of describing service orchestration have been released and used in the past, for example WS-BPEL [20]. WS-BPEL provides indeed many of the features necessary to describe workflows of services, plus communication aspects (ports, interfaces). Dynamic workflow reconfiguration can be expressed too [14]. However, WS-BPEL has been designed for high-level orchestration, while programming the internal logic of a single microservice requires fine-grained procedural constructs.

Our research team has been deeply involved in the microservice community and actively contributed to its broader adoption. As an open source project, Jolie has already built a community of developers worldwide - both in industry and in academia - taking care of the development, continuously improving its usability, and therefore broadening the adoption. Recent developments and contributions from our team are: extension of the type system [21], development of static type checking [24], addition of more iterative control structures to support programming, and inline automatic documentation [1]. These works geared up the development environment, and started the process of transforming it into a

full suite that makes the entire concept attractive to developers and marketable to companies.

The future is certainly not challenge-free. Security of the paradigm is an issue almost fully untouched [9]. Commercial-level quality packages for development are still far to come, despite the acceleration in the interest regarding the matter. Fully-verified software is an open problem the same way it is for more traditional development models. A main open problem is how microservices may integrate with the two main emerging platforms, which will likely dominate the near future: the cloud and the Internet of Things. While microservices seem ideal to run on the cloud, thanks to their properties of portability and elasticity, running on the Internet of Things still present some difficulties. In particular, many things have low computational capabilities and present higher risks from a security point of view, since they are easier to compromise [8]. As an example of this second point just consider that botnets such as Mirai [16] are composed by things (routers, IP cameras, digital video recorders, ...) which normally have very low protection (e.g., passwords fixed by the manufacturer and never changed) [6]. Hence integration of microservices and the Internet of Things would make the need for specific security solutions even more urgent.

## References

1. Bandura, A., Kurilenko, N., Mazzara, M., Rivera, V., Safina, L., Tchitchigin, A.: Jolie community on the rise. In: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), pp. 40–43 (2016)
2. Bondi, A.B.: Characteristics of scalability and their impact on performance. In: WOSP, pp. 195–203 (2000)
3. Tseitlin, A., Bennett, C.: Chaos Monkey Released Into The Wild (2012). <http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html>
4. Guidi, C., Lanese, I., Mazzara, M., Montesi, F.: Microservices: a language-based approach. *Present and Ulterior Software Engineering*, pp. 217–225. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-67425-4\\_13](https://doi.org/10.1007/978-3-319-67425-4_13)
5. de Almeida, E.S., Alvaro, A., Lucrédio, D., Garcia, V.C., de Lemos Meira, S.R.: Rise project: towards a robust framework for software reuse. In: IRI, pp. 48–53 (2004)
6. De Donno, M., Dragoni, N., Giaretta, A., Mazzara, M.: AntibiOTic: protecting IoT devices against DDoS attacks. In: *Proceedings of 5th International Conference in Software Engineering for Defence Applications* (2017)
7. Dragoni, N., Dustdar, S., Larse, S.T., Mazzara, M.: Microservices: Migration of a mission critical system (2017). <https://arxiv.org/abs/1704.04173>
8. Dragoni, N., Giaretta, A., Mazzara, M.: The internet of hackable things. In: *Proceedings of 5th International Conference in Software Engineering for Defence Applications* (2017)
9. Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, pp. 195–216. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-67425-4\\_12](https://doi.org/10.1007/978-3-319-67425-4_12)
10. McGarr, M., Bukoski, E., Moyles, B.: How We Build Code at Netflix (2016). <http://techblog.netflix.com/2016/03/how-we-build-code-at-netflix.html>

11. Evans, E.: *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Boston (2004)
12. Guidi, C., Lanese, I., Montesi, F., Zavattaro, G.: Dynamic error handling in service oriented applications. *Fundam. Inform.* **95**(1), 73–102 (2009)
13. Maurer, F., Succi, G., Holz, H., Kötting, B., Goldmann, S., Dellen, B.: Software process support over the internet. In: *Proceedings of the 21st international conference on Software engineering*, pp. 642–645. ACM (1999)
14. Mazzara, M., Abouzaid, F., Dragoni, N., Bhattacharyya, A.: Design, modelling and analysis of a workflow reconfiguration. In: *International Workshop on Petri Nets and Software Engineering*, pp. 10–24 (2011)
15. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. *Linux J.* **2014**(239), 2 (2014)
16. Mirai Botnet - wikipedia. [https://en.wikipedia.org/wiki/Mirai\\_\(malware\)](https://en.wikipedia.org/wiki/Mirai_(malware))
17. Montesi, F., Guidi, C., Zavattaro, G.: *Service-oriented programming with Jolie*. Web Services Foundations, pp. 81–107. Springer, New York (2014)
18. Neuman, B.C.: *Scale in distributed systems*. In: *Readings in Distributed Computing Systems*, pp. 463–489. IEEE Computer Society Press (1994)
19. Newman, S.: *Building Microservices*. O’Reilly Media Inc., Sebastopol (2015)
20. OASIS. *Web Services Business Process Execution Language Version 2.0* (2007). <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf>
21. Safina, L., Mazzara, M., Montesi, F., Rivera, V.: Data-driven workflows for microservices (genericity in Jolie). In *AINA* (2016)
22. Salikhov, D., Khanda, K., Gusmanov, K., Mazzara, M., Mavridis, N.: Jolie good buildings: Internet of things for smart building infrastructure supporting concurrent apps utilizing distributed microservices. In: *CCIT*, pp. 48–53 (2016)
23. Salikhov, D., Khanda, K., Gusmanov, K., Mazzara, M., Mavridis, N.: Microservice-based IOT for smart buildings. In: *WAINA* (2017)
24. Tchitchigin, A., Safina, L., Mazzara, M., Elwakil, M., Montesi, F., Rivera, V.: Refinement types in Jolie. In: *Spring/Summer Young Researchers Colloquium on Software Engineering, SYRCoSE* (2016)

# Static Binary Code Instrumentation for ARM Architecture

Mikhail Ermakov<sup>(✉)</sup>

Institute for System Programming of the Russian Academy of Sciences,  
25, Alexander Solzhenitsyn st., Moscow 109004, Russian Federation

[mermakov@ispras.ru](mailto:mermakov@ispras.ru)

<http://www.ispras.ru/en/groups/sp/>

**Abstract.** Binary code analysis is becoming a prominent technique in software development, covering tasks related to quality assurance and security. In this paper, we present an approach to static binary code instrumentation—persistent modification and extension of executable files—that we believe to be effective when used as a basis of implementation of various dynamic analysis techniques. We have developed an instrumentation framework targeting ARM ELF binary code that allows transforming files based on user-defined specifications. Specification language provides means to perform aspect-oriented programming targeting low-level groups of points in binary code to insert extra instrumentation code that can be supplied in C/C++ language. We have applied our framework to Avalanche—an automatic input generation tool based on dynamic symbolic execution—and achieved up to 10x increase in path traversal speed within a limited time frame compared to Valgrind-based dynamic binary instrumentation.

**Keywords:** Binary instrumentation · Dynamic analysis  
ARM architecture

## 1 Introduction

Program static and dynamic analysis tools have long become an important part of the software development process. Development teams employ powerful clusters and processing networks to automatically work on as many tasks as possible. The two main groups of analysis methods—static code analysis and dynamic program analysis—typically focus on software source and binary code respectively. Static analysis tools are generally incorporated in the actual development process (through IDE integration, nightly runs, etc.) and identify problematic sections of code that must be fixed as soon as possible. Dynamic analysis methods are designed to follow program execution. Excluding quality control through simple test execution, dynamic analysis might be performed as an independent stage of software development prior to important release dates. Such analysis sessions are designed to emulate user experience with the software product in order to identify critical issues to fix.

Dynamic analysis methods employ a broad range of approaches and techniques in order to emulate user experience and control program execution. These include system-wide monitoring and tracing, emulation and virtualization, and finally code instrumentation. In this paper we will focus on code instrumentation methods.

Code instrumentation implies the process of modifying program code to change the existing functionality and include additional functionality. Dynamic analysis methods typically favor only the latter approach to ensure reasonable and useful results. The additional code inserted in the target program is designed to produce traces or link with monitoring or debugging tools during execution. Generated information about program execution is processed in an intelligent way dependent on analysis goals.

It is possible to perform code instrumentation while working with various program representations.

- **source code instrumentation** allows incorporating new functionality while working with high-level semantic constructs. This approach does not require additional work with code generation—everything is performed by the existing build system. The approach requires the access to the source code and the build system, and thus cannot cover external and dynamically-generated code;
- **compiler-level instrumentation** [1] provides a powerful balance between high- and medium-level program code structure. This approach makes it easier to match target instrumentation points, control code modification and allows direct integration with optimization subsystems. As source code instrumentation, this approach can only be used to process available code; additionally, it imposes limitations on the build system;
- **dynamic binary instrumentation** provides means to process program execution thoroughly, modifying its code on the fly. As the process of instrumentation is directly tied to the program state and resource pool, it can be easily reconfigured, reversed or extended for certain portions of code. The drawbacks of dynamic instrumentation include increased influence on the execution state and the need to perform code parsing and modification for every program run.
- **static binary instrumentation** offers a range of analysis possibilities similar to source code and compiler-level instrumentation, but is applicable in cases where only binary files are available. It is less flexible than dynamic instrumentation in terms of code coverage and configuration, but offers better performance and ease-of-use prospects when used in large-scale analysis involving various user-end configurations and multiple program runs.

Currently, one of the large-scale analysis applications—automatic test generation for defect detection using program path traversal and symbolic execution techniques—features heavily in academic research and commercial use. It involves heavy processing of application code and requires significant amount of resources to obtain acceptable results. Thus, any practical improvement in code processing efficiency is highly desirable.

To this end, we have developed and implemented an approach to static instrumentation of binary code. We believe that persistent code modification and ease of deployment will provide practical benefits—decreasing execution overhead and making it easier to incorporate rapidly developing code analysis techniques. We choose to focus on binary code (and ARM architecture in particular) as it seems to be the primary focus of defect and vulnerability analysis and other applications in the field of software security.

The approach to code instrumentation and instrumentation frameworks were previously presented in [2,3]. This paper extends the publications with a description of code processing and modification algorithms. In Sect. 2 we give a brief overview of the instrumentation stages and identify the key features of instrumentation specifications, target binary code parsing and instrumentation code generation. Section 3 provides an overview of ELF section restructuring performed during instrumentation. Section 4 contains in-depth description of code transformation and optimization techniques. In Sect. 5 we showcase and discuss the results of practical experiments. The concluding section provides a brief evaluation of the presented work and outlines prominent future work directions.

## 2 Instrumentation Framework Overview

The work on a standalone static binary instrumentation framework for ARM architecture on Linux platforms was prompted by two main reasons. Firstly, existing instrumentation frameworks either do not target ARM architecture [4,11], provide support only in dynamic mode [5,6,8,13] or are distributed on a commercial basis [10]. Our focus in program analysis is strongly targeted at mobile platforms and Android and Tizen systems in particular. As both of these platforms are Linux-based we chose to process ELF executable format. This format describes a relatively formal yet extensible structure that is easy to process and includes a set of elements related to code organization and supported by the majority of compiler infrastructures and dynamic linkers.

Additionally, while existing instrumentation frameworks feature extensive API for code processing, they nevertheless require a certain amount of effort to design analysis tools. Our focus in program analysis targeting low-level instruction and basic block processing favored a more straightforward approach in creating code modification tool specifications—a simple aspect-oriented language that allows to extract necessary information from processed code to form execution traces.

Implementation-wise our framework is similar to PEBIL [4]—the main differences arise in the code generation process and the structure of instrumentation code. ARM instruction set is easier to work with due to reduced variation of instruction size compared to x86/x64 instruction sets and we are thus able to avoid main difficulties with instruction padding that the authors of PEBIL had to work with.

## 2.1 Input Specifications

The general instrumentation scheme follows input specification considerations described before. Our specification language allows to describe a set of instrumentation targets in the form of quadruples  $\langle T, C, F, D \rangle$ , where:

- $T$  identifies the instrumentation target type—instruction semantic group (i.e. arithmetic operation, memory access operation, etc.) or positional group (i.e. function entry point, basic block entry point, etc.)
- $C$  identifies instrumentation code—extra code that needs to be inserted in binary files. Instrumentation code is specified as a single block of C/C++ code, which might include calls to external libraries and a set of macro definitions that are processed by the instrumentation engine and serve as basic API.
- $F$  identifies source code filters that allow to limit code modification to specific functions (derived from source code through debugging information and symbol tables) present in binary files.
- $D$  identifies a set of external dependencies that allow to define, which external libraries need to be made accessible to binary files in order to satisfy the limitations imposed by the dynamic linker.

Within a specification file every instrumentation target is designed to cover specific individual instructions in binary code, which match the conditions limited by target type. Every target-instruction match will spawn a block of code by expanding macro definitions in the corresponding instrumentation code block using relevant instruction characteristics. Binary file instrumentation is limited to incorporating these blocks of code and modifying the original code control flow in such a way as to tie instruction execution with instrumentation code execution.

## 2.2 Instrumentation Code Generation

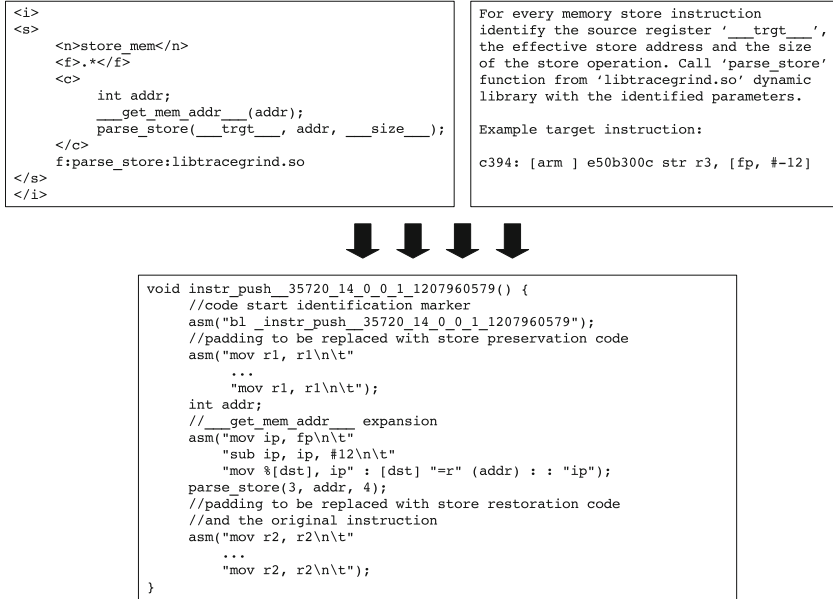
Instrumentation code generation is performed in a straightforward fashion and is directly tied to the binary code disassembly process based on linear algorithms. For every disassembled instruction we extract a set of properties that are later used to expand macro definitions in instrumentation code, identifying necessary padding size for register state save/restore operations and necessary instruction transformations. The latter two groups of properties are required to preserve original program functionality while inserting additional code.

We perform code disassembly and processing on a per-function basis. For this end in our current implementation we are dependent on the presence of several ELF sections not directly related to program execution—such as the symbol table—to identify blocks of instructions composing different functions, non-executable raw data blocks and instruction set transitions (ARM base instruction set to Thumb-2 set and vice versa).

After a function in binary file is fully disassembled every instruction is matched against each instrumentation target provided in specifications. For

every successful match instrumentation code is parametrized and added as a standalone function with a unique name in a source code file. Based on target type and instruction characteristics the actual block of instrumentation code is padded with inline assembly blocks. These blocks are later transformed into state preservation and restoration instructions, which guarantee that instrumentation code will not invalidate the execution state.

Figure 1 depicts an example instrumentation specification and the resulting instrumentation code generated for a matching instruction.



**Fig. 1.** Instrumentation code generation

Instrumentation code blocks corresponding to the same instructions or adjacent instructions in the original binary file are marked as “linked”. Linked blocks are later processed a specific way to optimize the number of control transfer instructions. The resulting source code file containing multiple instrumentation code functions is compiled into an object file by standard means. This file contains ELF-defined relocation information, which allows to identify individual instrumentation code blocks and references to external libraries. This information is typically used by linker tools; we employ the same data structures to incorporate instrumentation object files into fully formed binary files.

For basic instrumentation code specifications the generated object file can be easily appended to target binary files, which is enough to achieve the desired results (after control flow modifications described in Sect. 4 are performed)—a modified set of executable files that run additional code. However, if the



instrumentation code is dependent on external libraries for global variables or functions, more extensive modification must be performed in order to satisfy the dynamic linker contract. This modification is targeted at blocks of control data included in binary files during linking and compilation. Fortunately, with ELF format this data is stored in such a way that allows to perform necessary actions in a relatively straightforward way.

### 3 Working with ELF Files

The ELF format defines executable code files as a set of sections, which contain either raw data with its own semantics or a set of elements of specific nature. Sections have a set of attributes including their size, access flags and relative positions. Based on these attributes ELF sections are grouped in segments, which are loaded in virtual memory by the dynamic linker as solid blocks of data. Individual sections may incorporate information about other sections, which imposes certain limitations on how they can be modified (extended, moved within segments or moved to other segments). We have based our implementation on the analysis of common dynamic linkers used in Android and Tizen systems, as well as commonly used ELF section characteristics. We have constructed a list of ELF sections that might be freely modified or modified in a controllable fashion and a list of sections that were deemed too complex to process in a significant capacity. The second list includes executable and raw data sections, while the first list includes accessory sections.

The sections supporting the work of the dynamic linker (from the first list) include the following:

- `.dynamic`—section containing the most important data for the dynamic linker. Every external library used by the instrumentation code must be added as a reference item to this section, which causes its size to be increased;
- `.dynsym`, `.dynstr`—sections containing external symbol information (base properties and raw text specifying names);
- `.rel.plt`, `.rel.dyn`—sections containing information used to correct external references during executable code file initialization in virtual memory;
- `.got`, `.plt`—sections including active offsets and short blocks of code that are executed to perform control transfer to external libraries. As `.plt` section is heavily used by original code section of target file, it must be moved in a way to preserve relative offsets to the code section;
- `.gnu_version`, `.gnu_version_r`—sections containing versioning information for external libraries and symbols exported by them.

When instrumentation object file is generated, every reference to an external function will form a block of instructions that does not function properly (leads to an infinite loop or performs a possible illegal memory access). Appending instrumentation code to target executable file therefore requires a set of modifications that will correct these blocks of instructions. These modifications are directly tied to the sections mentioned in the list above. Every global variable

(function) referenced in the instrumentation code must have a corresponding entry in `.dynsym`, `.dynstr`, `.rel.dyn`, `.got` and `.gnu_version` sections (and `.plt`). These entries might already be present if the corresponding external symbols are used in the original code sections. If these entries are absent, they must be formed and inserted in all the sections; however, this causes an increase of section sizes. In turn, this disrupts the existing relative section placement within segments because of the data overlap. In order to fix the resulting code file, the segment map must be rebuilt. During the segment map rebuild sections from the second list containing raw data and executable code must keep their placement in regards to sections from the first list (particularly `.plt` and `.got`).

Given instrumentation specifications and information in target binary files we perform preliminary calculation of all necessary changes to section sizes. The resulting section placement is identified by checking every possible permutation of a section set and selecting the one that has the minimum overall size. The segment map is rebuild following the chosen permutation and the resulting binary file is processed in order to introduce basic changes to entries in the sections that were moved (`.rel.plt` in particular). These changes ensure that the modified sections hold entries relevant to the new segment map.

Figure 2 depicts an example of section configuration (commonly produced by Android/Tizen compiler toolchains; several sections irrelevant to the process are omitted) and modifications that are applied to it during instrumentation.

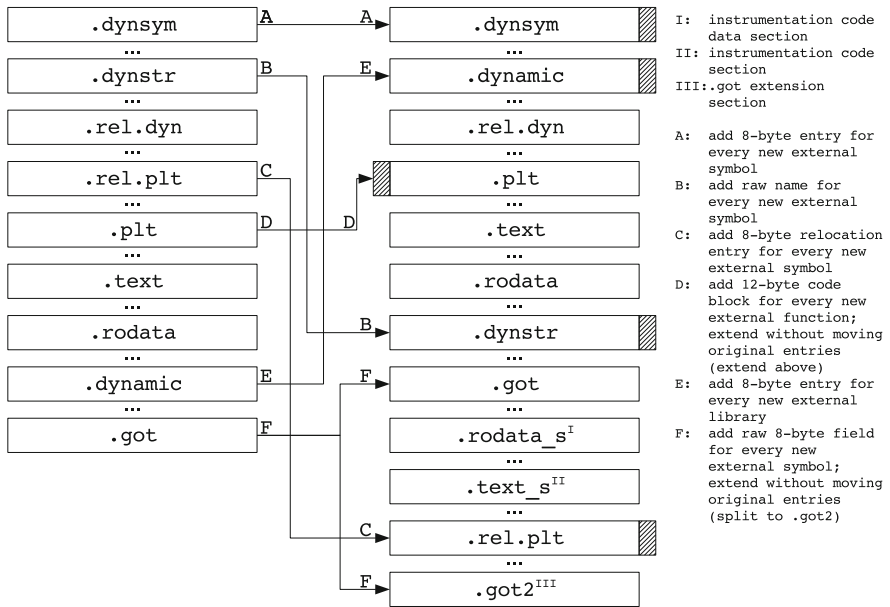


Fig. 2. ARM ELF section manipulation

## 4 Inserting Instrumentation Code

Altering control flow in the target binary code file is performed in a straightforward fashion. An instruction or a block of instructions at instrumentation point is replaced with an unconditional jump instruction to the corresponding block of instrumentation code. The replaced instructions are appended to the end of instrumentation code in order to preserve the original functionality. Finally, an unconditional jump instruction to return to the spot after the instrumentation point is appended after the replaced instructions.

For the majority of ARM and Thumb-2 instructions such modifications are safe in regards to their functionality. However, instructions that depend on the value of instruction pointer register or form a specific block with adjacent instructions require additional processing. We have introduced a set of rules to cover three groups of instructions that are relevant to this problem:

- relative branch instructions explicitly use the value of the instruction pointer register to calculate the target of the control flow jump; relocating these instructions to the instrumentation code is performed along with relative offset modification;
- arithmetic and memory instructions that explicitly use the value of the instruction pointer register are transformed into blocks of instructions that account for the difference between the instrumentation point and the start of instrumentation code block;
- conditional execution blocks from the Thumb-2 instruction set (IT blocks) must be relocated as a combined structure and duplicated over several instrumentation code blocks as the Thumb-2 reference imposes a set of rules over the composition of every IT block; the duplication is necessary to avoid in-depth processing of instruction semantics (in particular, flag register modification rules).

Inserting two jump instructions for every instrumentation point results in a notable overhead during program execution. While modifying program control flow is unavoidable during instrumentation unless the additional code is inserted directly into the original code sections (such modifications are complex and require a high level of disassembly and code analysis not present in existing static binary instrumentation frameworks), we nevertheless aim to decrease the number of control switches whenever possible. To this end we have implemented two optimization mechanisms:

- instrumentation code block “linking” allows us to insert jump instructions to reach the beginning of the next instrumentation code block after the end of the previous one in case these blocks correspond to two adjacent instrumentation points; in this scenario there is no need to perform two consecutive control flow switches without any meaningful instructions in between;
- instrumentation point extension allows us to relocate a linear sequence of instructions starting at instrumentation point and ending at the next instrumentation point to the end of instrumentation block; the functionality is not

changed (since moved instructions are processed according to the rules above) and we are able to use “linking” to remove redundant jump instructions; this optimization is only available for actual linear sequences within a single basic block.

#### 4.1 Code Linkage

During final code instrumentation stages we also perform actions that are normally carried out by a linker within a compiler infrastructure. We use relocation and other supplementary information stored in the instrumentation object file to perform the following modifications in the instrumentation code appended to the target binary file:

- correcting offsets from instrumentation code section to instrumentation data section based on their relative placement in the target file;
- correcting offsets from instrumentation code to the `.plt` and `.got` sections in order to correctly call functions and use global variables from external libraries; these offsets might either point to existing entries in these sections (already defined in the original version of the binary file) or to the new entries added during the stage described in Sect. 3.

## 5 Practical Evaluation

We have implemented our instrumentation framework on top of the open source package `binutils` [12] (for ARM ELF processing and modification) and `gcc` [7] for automatic code generation.

We have conducted several sets of experiments with the framework in order to identify the limitations of its use and overhead considerations. Our main focus of research was connected with iterative dynamic analysis and dynamic symbolic execution where we attempted to use the advantages of static instrumentation to decrease the overhead over multiple runs of application under analysis. To test the viability of static binary instrumentation we have chosen `Avalanche` [9]—an analysis framework for generating input data and program path traversal. The original version of `Avalanche` employed dynamic binary instrumentation to perform two different operations—light-weight basic block coverage analysis for program path prioritization, and heavy-weight data flow tracking and path predicate generation.

We have designed our own versions of instrumentation tools for these tasks using the implemented instrumentation framework. Our basic block coverage checker uses small code snippets in conjunction with a simple hash library. In addition the code snippets include blocks of self-modifying code that is designed to revert control flow modifications performed during instrumentation—after a basic block entry point is added to the set during program execution the original instructions at the instrumentation point are restored. Every consecutive execution of this basic block is performed with zero overhead. This optimization

is comparable to the original Avalanche implementation that used the Valgrind framework.

Our data flow tracker and path predicate generation tool is designed as an external library and optimized for the ARM architecture instructions. It is mostly identical to the original Avalanche implementation that used the Valgrind framework.

Applying our instrumentation plugins to a set of applications allowed us to achieve a significant increase in analysis efficiency without losing its precision in a given time frame (see Table 1). The decreased time of instrumented program runs allowed to process more execution paths and uncover additional critical defects (invalid memory accesses causing segmentation faults) for two programs from the set.

**Table 1.** Static instrumentation vs. dynamic instrumentation for Avalanche

Target program	TGT (dynamic)	TGT (static)	CCT (dynamic)	CCT (static)	I/D (dynamic)	I/D (static)
cjpeg	4.12	0.33	1.24	0.32	4009/1	4163/1
djpeg	3.49	0.16	1.38	0.09	3455/0	36443/0
mpeg2dec	3.73	0.21	1.28	0.09	4051/1	13459/1
mpeg3dump	10.23	0.6	1.58	0.1	2637/2	37528/2
swfdump	4.61	0.48	1.14	0.33	3528/1	14759/4
qtdump	6.01	0.35	2.75	0.11	1592/2	35402/3

*TGT*—average trace generation time (seconds).

*CCT*—average coverage check time (seconds).

*I/D*—inputs checked/defects detected in the 2 h frame.

For a single executable (cjpeg) the use of static binary instrumentation caused a significant shift in the distribution of plugin work time, bringing the solver component (used to generate new input data from path constraints) to the forefront. Due to minor differences in basic block calculation Avalanche under static instrumentation hit a seemingly prominent execution path subtree far from the entry point. In turn, this resulted in larger path constraints being generated—thus, the solver component took more time to process the constraints and was able to generate fewer new inputs.

## 6 Conclusion

In this paper we have presented an approach to perform static binary instrumentation for executable code files in ARM ELF format. We have implemented an instrumentation framework that allows to design low-level code modification tools with an easy-to-use aspect-like specification language.

While we believe our framework to be useful for various research and practical applications, there are multiple directions for future work. Firstly, we want to

increase the level of analysis performed during disassembly and code generation in order to obtain more information. This information can be used to optimize state preservation and restoration blocks that we include in the instrumentation code (in particular, flag and register liveness analysis). Secondly, while binutils disassembly algorithms are fairly reliable, they lose precision when targeted at obfuscated code or code stripped of any information that is not related to its execution. Therefore, integrating the instrumentation framework with more complex and powerful disassembly tool appears to be a relevant and desirable improvement. Finally, we wish to improve the instrumentation framework API and instruction set coverage (ARMv8 support, ARMv7 vector and floating point instruction support).

## References

1. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: a fast address sanity checker. In: USENIX Annual Technical Conference (2012)
2. Ermakov, M.K., Vartanov, S.P.: Dynamic analysis of ARM ELF shared libraries using static binary instrumentation. *Trudy ISP RAN/Proc. ISP RAS* **27**(1), 5–24 (2015)
3. Ermakov, M.K.: Dynamic analysis of ARM ELF executable code using static binary instrumentation. *SPbSPU J. Comput. Sci. Telecommun. Control Syst.* **1**(236), 108–117 (2016)
4. Laurenzano, M., Tikir, M., Carrington, L., Snavely, A.: PEBIL: efficient static binary instrumentation for Linux. In: ISPASS, pp. 175–183. IEEE Computer Society (2010)
5. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE 2011), pp. 9–16. ACM, New York (2011)
6. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), pp. 190–200. ACM, New York (2005)
7. GCC project home page. <https://gcc.gnu.org/>
8. Bruening, D.L.: Efficient, Transparent, and Comprehensive Runtime Code Manipulation. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. AAI0807735 (2004)
9. Isaev, I.K., Sidorov, D.V.: The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. *Program. Comput. Softw.* **36**(4), 225–236 (2010)
10. Anand, K., Smithson, M., Elwazeer, K., Kotha, A., Gruen, J., Giles, N., Barua, R.: A compiler-level intermediate representation based binary analysis and rewriting system. In: Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys 2013), pp. 295–308. ACM, New York (2013)
11. Zhang, M., Qiao, R., Hasabnis, N., Sekar, R.: A platform for secure static binary instrumentation. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE 2014), pp. 129–140. ACM, New York (2014)

12. Binutils project home page. <https://www.gnu.org/software/binutils/>
13. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007), pp. 89–100. ACM, New York (2007)

# A Behavioural Theory for Reflective Sequential Algorithms

Flavio Ferrarotti<sup>(✉)</sup>, Klaus-Dieter Schewe, and Loredana Tec

Software Competence Center Hagenberg, Hagenberg, Austria  
flavio.ferrarotti@scch.at, kd.schewe@gmail.com, loredana.tec@gmail.com

**Abstract.** We develop a behavioural theory of reflective sequential algorithms (RSAs), i.e. algorithms that can modify their own behaviour. The theory comprises a set of language-independent postulates characterising the class of RSAs, an abstract machine model that provably satisfies the postulates, and a proof that all RSAs are captured by this machine model. As in Gurevich's thesis for sequential algorithms RSAs are sequential-time, bounded parallel algorithms, where the bound depends on the algorithm only and not on the input. Different from the class of sequential algorithms every state of an RSA includes a representation of the algorithm in that state, thus enabling linguistic reflection. The model of reflective Abstract State Machines (rASMs) extends sequential ASMs using extended states that include an updatable representation of the main ASM rule to be executed by the machine in that state.

## 1 Introduction

Self-adaptive systems have recently attracted a lot of interest in research, in particular in connection with systems of (cyber-physical) systems [9]. Adaptivity refers to the ability of a system to change its own behaviour. In the context of programming this concept, known under the term *linguistic reflection*, appears already in LISP [11], where programs and data are both represented uniformly as lists, and thus programs represented as data can be executed dynamically by means of an evaluation operator. Run-time and compile-time linguistic reflection in programming and database research has been investigated in general by Stemple, Van den Bussche and others in [12, 13].

This raises the questions how the development of adaptive systems can be supported by state-based rigorous methods such Abstract State Machines (ASMs) [5]. These methods are coupled with a genericity promise, i.e. they can be applied universally to a large class of systems supporting rigorous specification on any level of abstraction, seamless step-wise refinement from high-levels of abstraction down to implemented code, validation and tracing of requirements

---

The research reported in this paper results from the projects *Behavioural Theory and Logics for Distributed Adaptive Systems* and *Higher-Order Logics and Structure* supported by the **Austrian Science Fund (FWF: [P26452-N15] & [I2420-N31])**.



through the refinement process, and verification of specifications against the requirements on grounds of dedicated logics. However, reflective algorithms are not yet covered.

Gurevich's celebrated *sequential ASM thesis* [8] states that sequential algorithms are captured by sequential ASMs. A key contribution of this thesis is the language-independent characterisation of a reflective algorithm by a small set of intuitively understandable *postulates*, by means of which a definition of a sequential algorithm on an arbitrary level of abstraction is given. Then it can be shown that every sequential algorithm as stipulated by the postulates can be step-by-step simulated by a sequential ASM.

Based on this thesis the notion *behavioural theory* has been introduced for a triplet comprising (1) a set of *postulates* that characterises a class of algorithms or systems, (2) an abstract machine model together with a *plausibility* proof that the abstract machine satisfy the postulates, and (3) a *characterisation* proof that all algorithms stipulated by the postulates are captured by the abstract machine model. That is, the sequential ASM thesis provides the behavioural theory of sequential algorithms. Other examples cover the behavioural theory of parallel algorithms developed by Blass and Gurevich [1,2], its simplification by Ferrarotti et al. [6] using a different set of postulates, the behavioural theory of concurrent algorithms [4], and the behavioural theory of non-deterministic database transformations [10].

In this paper we investigate a behavioural theory for reflective, sequential algorithms, which was conjectured in [7]. In light of the significantly increased technical difficulties that have to be addressed when unbounded parallelism is permitted (compare the proofs in [6] with those in the sequential ASM thesis [8]) we first restrict the emphasis on sequential algorithms, where parallelism is a priori bound and does not depend on the state.

We first develop a set of postulates characterising *reflective sequential algorithms* (RSAs). The key issue is that an RSA must have some representation of itself, but this has to be left completely abstract. We argue that this is possible, which leads to extended states, where abstract terms that appear in the description of the algorithm are used as values, which requires a distinction concerning their interpretation in a state. The tricky problem is the generalisation of bounded exploration, for it is clear that all means of an algorithm to change itself must appear somehow in the algorithm's description. We argue that there is still a bounded exploration witness, i.e. a set of ground terms that determines the update sets yielded in a state, but the bounded exploration postulate will nonetheless require some sophisticated differentiation concerning the interpretation of terms. The postulates for RSAs will be discussed in Sect. 2.

In Sect. 3 we proceed with the definition of reflective sequential ASMs (rASMs), which will be a straightforward extension of ASMs using a dedicated location *self* capturing the (syntax of the) sequential ASM that is to be applied in this state. This determines the runs of a rASM with the difference that in each step now a possibly different ASM may have been used to determine the

updates that mark the state changes. We also briefly sketch the plausibility theorem though it commonly addresses the simpler proof direction.

Section 4 addresses the proof of the characterisation theorem, which is again accomplished by a sequence of lemmata, the key problem being that there is a theoretically unbounded number of different algorithms that nonetheless have to be handled uniformly. This section will be the technical key contribution of this paper. We conclude with a brief summary and outlook in Sect. 5.

## 2 Reflective Algorithms and Their Axiomatisation

The celebrated sequential ASM thesis needs only three simple, intuitive postulates to define sequential algorithms (for details see the deep discussion in [8]):

**Sequential time:** Each sequential computation proceeds by means of a *transition function*  $\tau : \mathcal{S} \rightarrow \mathcal{S}$ , which maps a state  $S \in \mathcal{S}$  to its successor state  $\tau(S)$ .

**Abstract state:** Each state  $S \in \mathcal{S}$  is a *Tarski structure* defined over a signature  $\Sigma$ , i.e. a set of function symbols, by means of interpretation in a base set  $B_S$ . States, initial states and transitions are closed under isomorphisms.

**Bounded exploration:** There is a fixed, finite set of ground terms  $W$  called *bounded exploration witness* such that whenever two states coincide on  $W$ , the update sets that determine the changes in the transition to the respective successor states are equal.

The postulates imply that sequential algorithms can only check agreement between states on a *fixed* and *finite* set of *ground terms* (i.e., the bounded exploration witness in the bounded exploration postulate for sequential algorithms). Reflective algorithms, however, do not satisfy this principle, as the following simple Example 1 shows. The RSA in the example does *not* satisfy the bounded exploration postulate for sequential algorithms. However, it is NOT the question, whether a different, non-reflective algorithm exists that solves the same problem, but whether such an algorithm would also be *behaviourally equivalent*.

*Example 1.* We describe a RSA that takes as input a search term  $t$ , a perfect binary tree  $T$ , i.e., a binary tree in which all interior nodes have two children and all leaves have the same depth, and a function *label* which maps the set of nodes to an arbitrary set of labels. It traverses the graph in a breadth first order starting by its root  $r$ . If the term  $t$  appears as label of some node in the input tree, then the algorithm updates *result* to the lowest level in the tree which contains a node labeled with  $t$ .

We assume that in every initial state  $level = 0$ ,  $currentNode = r$  and  $result = undef$ . Let *cond* be the following function from the natural numbers to Boolean terms:

$$\begin{aligned}
\text{cond}(0):: &= \text{label}(r) = t \\
\text{cond}(1):: &= \text{label}(\text{leftChild}(r)) = t \vee \text{label}(\text{rightChild}(r)) = t \\
\text{cond}(2):: &= \text{label}(\text{leftChild}(\text{leftChild}(r))) = t \vee \text{label}(\text{rightChild}(\text{leftChild}(r))) = t \vee \\
&\quad \text{label}(\text{leftChild}(\text{rightChild}(r))) = t \vee \text{label}(\text{rightChild}(\text{rightChild}(r))) = t \\
&\quad \vdots \quad \quad \quad \vdots \\
\text{cond}(n):: &= \underbrace{\text{label}(\text{leftChild}(\dots \text{leftChild}(r)\dots)) = t \vee \dots}_{\dots \vee \underbrace{\text{label}(\text{rightChild}(\dots \text{rightChild}(r)\dots)) = t}_{2^n}}.
\end{aligned}$$

The algorithm works as follows:

```

1: if  $\text{currentNode} \neq \text{undef} \wedge \text{result} = \text{undef}$  then
2:   if  $\text{label}(r) = t$  then
3:      $\text{result} := \text{level}$ 
4:   else
5:      $\text{currentNode} := \text{leftChild}(\text{currentNode})$ 
6:      $\text{level} := \text{level} + 1$ 
7:     Replace the Boolean term in the if-statement in line 2 by the
       interpretation of  $\text{cond}(\text{level}+1)$ 
8:   endif
9: endif

```

Notice that during a run or computation of this algorithm not only the state of the algorithm evolves, but also the algorithm itself. In fact, the Boolean term in the if-statement in line 2 changes with every state transition until either the searched term  $t$  is found or all the levels of the input tree have been exhausted.

As we consider input trees of arbitrary size, this means that there is no fixed and finite bounded exploration witness for this algorithm, as we would need to either include all Boolean terms in the infinite set  $\{\text{cond}(0), \text{cond}(1), \dots\}$ , or include a different Boolean term  $\text{cond}(\text{level} + 1)$  depending on the interpretation of  $\text{level} + 1$  in the current state.  $\square$

## 2.1 Reflective Sequential Time Postulate

Clearly, when extending the notion of sequential algorithm to include reflection we think of pairs  $(S_i, P_j)$  comprising a state  $S_i$  (as in the sequential thesis), and a sequential algorithm  $P_j$ . Thus, we can consider transition functions  $\tau_j : (S_i, P_j) \mapsto (S_{i+1}, P_j)$  without changing the sequential algorithm  $P_j$ . Likewise we may consider transition functions  $\sigma_i : (S_i, P_j) \mapsto (S_i, P_{j+1})$  changing only the algorithm. In general, a transition of a RSA can then involve both: updates to the state and updates to the algorithm.

**Postulate 1 (Reflective Sequential Time Postulate).** Let  $\mathcal{S}_P$  and  $\mathcal{I}_P$  denote the set of states and initial states of a sequential algorithm  $P$ , respectively ( $\mathcal{I}_P \subseteq \mathcal{S}_P$ ). A RSA  $\mathcal{A}$  consists of the following:

- A non-empty set  $\mathcal{P}_A$  of *sequential algorithms*;
- An *initial algorithm*  $P_0 \in \mathcal{P}_A$ ;
- A non-empty set  $\mathcal{S}_A = \bigcup_{P_i \in \mathcal{P}_A} \mathcal{S}_{P_i}$  of *states*;
- A non-empty set  $\mathcal{I}_A = \mathcal{I}_{P_0} \subseteq \mathcal{S}_A$  of *initial states*;
- A set of extended-states  $\mathcal{E}_A = \mathcal{S}_A \times \mathcal{P}_A$ ;
- A *one-step transformation* function  $\tau_A : \mathcal{E}_A \rightarrow \mathcal{E}_A$  such that  $\tau_A((S, P)) = (S', P')$  only if  $\tau_P(S) = S'$  for the one-step transformation function  $\tau_P$  of the sequential algorithm  $P$ .

Then a *run* or *computation* of a reflective algorithm corresponds to a sequence of pairs  $(S_0, P_0), (S_1, P_1), (S_2, P_2), \dots$ , where  $S_0$  is an initial state in  $\mathcal{I}_A$ ,  $P_0$  is the initial algorithm, and  $(S_{i+1}, P_{i+1}) = \tau_A((S_i, P_i))$  holds for every  $i \geq 0$ .

This leads to the following three fundamental questions which we try to answer in the remaining part of this section:

- Q1.** How can we finitely represent a sequential algorithm  $P$  without having to adopt a concrete language for the specification of  $P$ ?
- Q2.** How can we finitely characterise changes to the representation of the sequential algorithms in all states?
- Q3.** How can we define behavioural equivalence of RSAs independently from the representation of the sequential algorithms in each state?

## 2.2 Reflective Abstract Extended-State Postulate

Concerning Q1 we observe that according to the sequential ASM thesis it suffices to represent a sequential algorithm  $P$  by a set of pairs  $(S, \Delta(P, S))$  comprising a state  $S$  and the update set of  $P$  in that state. A consequence of the proof of the sequential ASM thesis in [8] is that update sets  $\Delta(P, S_i)$  ( $i = 1, 2$ ) are equal, if the states  $S_1$  and  $S_2$  are  $W$ -equivalent for a fixed bounded exploration witness  $W$ . We have  $S_1 \sim_W S_2$  iff  $E_{S_1} = E_{S_2}$ , where  $E_S$  is the equivalence relation on  $W$  defined by  $E_S(t_1, t_2) \equiv \text{val}_S(t_1) = \text{val}_S(t_2)$ <sup>1</sup>. It is therefore sufficient to replace the state  $S$  by a condition  $\varphi_{[S]}$ , which evaluates to true on states that are  $W$ -equivalent to  $S$ . As there can only be finitely many  $W$ -equivalence classes, we obtain an abstract finite representation by a finite set of pairs  $(\varphi_i, \Delta_i)$  ( $i = 1, \dots, k$ ).

Therefore, we conclude that we can capture the state-algorithm pairs in a RSA by an extension  $\Sigma_{ext}$  of the signature  $\Sigma$  using additional function symbols to represent the sequential algorithm, e.g. capturing in the state the signature of the algorithm as well as some syntactic description of it. For this, we must further permit new function symbols to be created, which can be done by exploiting the concept of “reserve”. We also conclude that the representation of algorithms in a state requires terms that are used by the algorithms to appear as values. So we have to allow terms over  $\Sigma$  (including the dormant function symbols in the reserve) to be at the same time values in an extended base set. In order

<sup>1</sup> As usual,  $\text{val}_S(t)$  denotes the interpretation of a ground term  $t$  as a value in the base set of a state  $S$ .

to distinguish the interpretation of such terms  $t$  as values  $val_{(S,P)}(t)$  of the base set of an extended-state  $(S, P)$  (which in any extended-state evaluate to themselves) from their interpretation as terms over  $\Sigma$ , we use  $raise_{(S,P)}(t)$  to denote the latter case. We may further assume that  $raise$  results in a proper term over  $\Sigma$  not containing any extra-logical constructs that are needed in the representation of an algorithm such as keywords.

**Postulate 2 (Reflective Abstract Extended-State Postulate).** Let  $\mathcal{A}$  be RSA. Fix a finite signature  $\Sigma_{algo}$  of function symbols so that every algorithm  $P \in \mathcal{P}_{\mathcal{A}}$  can be finitely represented as some first-order structure of signature  $\Sigma_{algo}$ .

- Every  $P$  in  $\mathcal{P}_{\mathcal{A}}$  is a first-order structure of signature  $\Sigma_{algo}$  which encodes a finite representation of a sequential algorithm.
- Every state  $S$  in  $\mathcal{S}_{\mathcal{A}}$  is a first-order structure of some signature  $\Sigma_S$  such that  $\Sigma_S \cap \Sigma_{algo} = \emptyset$ .
- Every extended-state  $(S, P)$  in  $\mathcal{E}_{\mathcal{A}}$  is a first-order structure of (extended) signature  $\Sigma_{ext} = \Sigma_S \cup \Sigma_{algo}$ .
- The one-step transformation function  $\tau_{\mathcal{A}}$  does not change the base set of any extended-state of  $\mathcal{A}$ .
- The sets  $\mathcal{S}_{\mathcal{A}}$  and  $\mathcal{I}_{\mathcal{A}}$  are closed under isomorphisms.
- If  $(S_1, P_1), (S_2, P_2) \in \mathcal{E}_{\mathcal{A}}$ ,  $S_1$  and  $S_2$  are isomorphic,  $P_1$  and  $P_2$  are behavioural equivalent sequential algorithms<sup>2</sup>, and further  $\tau_{\mathcal{A}}(S_1, P_1) = (S'_1, P'_1)$  and  $\tau_{\mathcal{A}}(S_2, P_2) = (S'_2, P'_2)$ , then also  $S'_1$  and  $S'_2$  are isomorphic and  $P'_1$  and  $P'_2$  are behavioural equivalent.

Same as in the sequential ASM thesis, we need some minimal *background of computation*. Therefore, for every extended state  $(S, P)$ , we assume that  $S$  includes a binary function “=” for equality, nullary functions **true**, **false** and **undef** with **true**  $\neq$  **false** and **true**  $\neq$  **undef**, the usual Boolean functions, the set of all ordered pairs, and an infinite reserve of elements. As explained before, we further assume that  $P_i$  includes, as values in its base set, the set of all possible ground terms (including the dormant function symbols in the reserve).

### 2.3 Reflective Bounded Exploration Postulate

Concerning Q2 the problem is that in general we must expect that each sequential algorithm  $P_i$  represented in an extended-state  $(S_i, P_i)$  has its own bounded exploration witness  $W_i$ . However, we know from the sequential ASM thesis that  $W_i$  is somehow contained in the finite representation of  $P_i$ . For instance, the sequential ASM rule constructed in the proof of the sequential ASM thesis only contains subterms of terms in  $W_i$ , and this holds analogously for any other representation of  $P_i$ . This implies that the terms in  $W_i$  result by interpretation from terms that appear in the representation of any sequential algorithm. Thus, there must exist a finite set of terms  $W$  such that its interpretation in an extended

<sup>2</sup> Two sequential algorithms  $P_1$  and  $P_2$  are *behavioural equivalent* if  $\mathcal{S}_{P_1} = \mathcal{S}_{P_2}$ ,  $\mathcal{I}_{P_1} = \mathcal{I}_{P_2}$  and  $\tau_{P_1} = \tau_{P_2}$ . Behavioural equivalent sequential algorithms have the same runs.

state yields both values and terms, and the latter represent  $W_i$ . We will continue to call  $W$  a *bounded exploration witness*. Consequently, the interpretation of  $W$  and of its interpretation in an extended state suffice to determine the update set in that state. This leads to our *bounded exploration postulate* for RSAs.

**Definition 1 (Strong Coincidence).** *Let  $(S, P)$  and  $(S', P')$  be extended-states of a RSA. Let  $W = W_{st} \cup W_{wt}$  be a set of ground terms. We say that  $(S, P)$  and  $(S', P')$  strongly coincide over  $W$  iff the following holds:*

- For every  $t \in W_{st}$ ,  $val_{(S,P)}(t) = val_{(S',P')}(t)$ .
- For every  $t \in W_{wt}$ ,
  1.  $val_{(S,P)}(t) = val_{(S',P')}(t)$ .
  2.  $val_{(S,P)}(raise_{(S,P)}(t)) = val_{(S',P')}(raise_{(S',P')}(t))$ .

In our third and last postulate we use  $\Delta(\mathcal{A}, (S, P))$  to denote the set of updates produced by a RSA  $\mathcal{A}$  in an extended-state  $(S, P)$ .

**Postulate 3 (Reflective Bounded Exploration Postulate).** For every RSA  $\mathcal{A}$ , there is a finite set  $W = W_{st} \cup W_{wt}$  of ground terms such that  $\Delta(\mathcal{A}, (S, P)) = \Delta(\mathcal{A}, (S', P'))$  whenever extended-states  $(S, P)$  and  $(S', P')$  of  $\mathcal{A}$  strongly coincide on  $W$ .

If a set of ground terms  $W = W_{st} \cup W_{wt}$  satisfies the reflective bounded exploration postulate, we call it a *reflective bounded exploration witness* (R-witness for short) for  $\mathcal{A}$ .

## 2.4 Reflective Sequential Algorithms and Behavioural Equivalence

Our three postulates give us the following machine independent definition of RSAs.

**Definition 2.** *A reflective sequential algorithm (RSA) is an algorithm satisfying the Reflective Sequential Time, Reflective Abstract State and Reflective Bounded Exploration Postulates.*

*Example 2.* Let us consider the algorithm in Example 1. The reflective sequential time and reflective abstract state postulates are clearly satisfied by this algorithm. Let

$$W_{st} = \{currentNode, undef, result, level, level + 1, leftChild(currentNode)\}$$

and  $W_{wt} = \{cond(level)\}$ . It is not difficult to see that if two extended-states coincide on  $W = W_{st} \cup W_{wt}$ , then the algorithm considered in this example produces the same set of updates in both extended-states. Thus, it also satisfies the reflective bounded exploration postulate, and consequently our definition of RSA.  $\square$

Next, we turn our attention to our final fundamental question Q3. The problem here is that the notion of behavioural equivalence of two sequential algorithms is bound to these having the same signature, on grounds of which we can request that the sets of runs must be identical. This cannot be carried over to RSAs in a straightforward way. However, we should be able to obtain a bijection between runs  $(S_0, P_0) \rightarrow (S_1, P_1) \rightarrow (S_2, P_2) \rightarrow \dots$  and  $(S'_0, P'_0) \rightarrow (S'_1, P'_1) \rightarrow (S'_2, P'_2) \rightarrow \dots$  for two RSAs  $\mathcal{A}$  and  $\mathcal{A}'$ . Then we should clearly have that  $S_i = S'_i$  holds for all  $i$ , and that  $P_i$  and  $P'_i$  are behaviourally equivalent as non-reflective, sequential algorithms. This is not yet satisfactory, as  $P_i$  and  $P'_i$  may still operate on different signatures.

We can argue that it is sufficient to consider the restrictions of  $P_i$  and  $P'_i$  on the “standard” part of the signatures, i.e. the functions that do not take terms as values. This would allow the algorithms  $P_i$  and  $P'_i$  to differ in their changes to themselves, but these differences have de facto no effect, as the updates yielded by these algorithms produce the same state transition and result in modified, yet behaviourally equivalent algorithms throughout the complete run. In other words, the possibly differing changes to the algorithm may extend the signature by functions or integrate fragments of “code” that are never used and thus have no effect on the updates.

**Definition 3 (Behavioural Equivalent RSAs).** *Let  $r_1 = (S_0, P_0), (S_1, P_1), (S_2, P_2), \dots$ , and  $r_2 = (S'_0, P'_0), (S'_1, P'_1), (S'_2, P'_2), \dots$ , be runs of RSAs. We consider that  $r_1$  and  $r_2$  are essentially equivalent runs if for every  $i \geq 0$  the following holds:*

1.  $S_i = S'_i$ .
2. The restrictions  $P_i|_{\Sigma}$  and  $P'_i|_{\Sigma}$  of, respectively,  $P_i$  and  $P'_i$  to the signature  $\Sigma$  of  $S_i$  and  $S'_i$ , constitute behavioural equivalent non-reflective sequential algorithms.

*Two RSAs  $\mathcal{A}$  and  $\mathcal{A}'$  are behavioural equivalent RSAs iff  $\mathcal{A}$  and  $\mathcal{A}'$  have essentially equivalent classes of essentially equivalent runs. More precisely, iff there is a bijection  $\zeta$  between runs of  $\mathcal{A}$  and  $\mathcal{A}'$ , respectively, such that  $r$  and  $\zeta(r)$  are essentially equivalent for all run  $r$ .*

### 3 Reflective Abstract State Machines

In this section we define a model of *reflective* ASMs (rASMs for short) and show that every rASM is a RSA in the precise sense of Definition 2. Given a signature  $\Sigma$ , i.e. a set of function symbols, then a sequential ASM-rule over  $\Sigma$  is defined as follows [5]:

**assignments.**  $f(t_1, \dots, t_{ar_f}) := t_0$  (with terms  $t_i$  built over  $\Sigma$ ) is a rule.

**branching.** If  $r_+$  and  $r_-$  are rules and  $\varphi$  is a Boolean term, then also **if**  $\varphi$  **then**  $r_+$  **else**  $r_-$  **endif** is a rule.

**bounded parallel composition.** If  $r_1, \dots, r_n$  are rules, then also **par**  $r_1 \dots r_n$  **endpar** is a rule.

Each rule can be interpreted in a state, and doing so yields an update set. In general, a *location* is a pair  $\ell = (f, (a_1, \dots, a_k))$  with a function symbol  $f \in \Sigma$  and a  $k$ -tuple ( $k$  being the arity of  $f$ ) of values from the fixed base set  $B$ , and an *update* is a pair  $(\ell, a_0)$  with a value  $a_0 \in B$ .

The rules of an rASM are also sequential ASM rules, and the interpretation of these rules in terms of update sets coincides with those of sequential ASMs as defined in [5]. The key difference is that rASMs work over extended-states, where each extended-state includes a finite representation of the rule that determines the update set produced by the machine in the current extended-state. In this way, we also allow an rASM to produce updates to its current rule.

Let  $(S, R)$  be an extended state of a rASM  $\mathcal{M}$ . We assume that the sub-structure  $S$  includes the following background of computation:

- An infinite reserve of values and function names.
- All ordered pairs of elements in the base set.
- The usual Boolean functions and usual constants **true**, **false** and **undef**.
- The “program” functions *update*, *par*, *if*.

The “program” functions are static and interpreted as follows:

- $update(f(t_1, \dots, t_n), t_0) = (t_0, t_1, \dots, t_n)$
- $par(t_1, t_2) = (val_S(t_1), val_S(t_2))$
- $if(t_1, t_2) = (t_1, val_S(t_2))$ .

Notice that the following function induces a one-to-one correspondence between ASM rules and “program” terms, so that every ASM rule can be represented as a “program” term.

- $progToFunction(f(t_1, \dots, t_n) := t_0) = update(f(t_1, \dots, t_n), t_0)$ .
- $progToFunction(\mathbf{if} \varphi \mathbf{then} R \mathbf{endif}) = if(\varphi, progToFunction(R))$ .
- $progToFunction(\mathbf{par} R_1 R_2 \mathbf{endpar}) = par(progToFunction(R_1), progToFunction(R_2))$ .

The sub-structure  $R$  of the extended-state  $(S, R)$  (i.e., the structure which contains the encoding of the “current” ASM rule) includes:

- The set of all ground terms.
- A distinguished location *self* interpreted as a “program” term (the current ASM rule).
- A finite alphabet  $A$  (the alphabet of the ground terms) and all strings in  $A^*$ .
- A constant  $s_i$  for each symbol  $s_i \in A$  and a constant  $\lambda$  for the empty string.
- The usual string manipulation functions, including the concatenation function “.”.
- A total injective function *TermToString* from the set of all terms of vocabulary  $\Sigma$  to  $A^*$ .
- A partial function *StringToTerm* defined as the inverse of *TermToString*.
- A function *argumentNo*( $t, n$ ) which returns the  $n$ -th argument of the term  $t$ .
- A function *insertArgument*( $s, n, t$ ) which returns a copy of  $t$  with its  $n$ -th argument replaced by  $s$ .



Since in each extended-state  $(S, R)$  of a rASM, the sub-structure  $R$  represents a uniquely determined sequential ASM rule, we usually refer to it as a rule rather than as a structure, meaning the rule corresponding to the “program term” in the location *self*.

**Definition 4.** An rASM  $\mathcal{M}$  is formed by:

- A non-empty set  $\mathcal{R}_{\mathcal{M}}$  of sequential ASM rules (represented as first-order structures).
- An initial rule  $R_0 \in \mathcal{R}_{\mathcal{M}}$ .
- A non-empty set  $\mathcal{S}_{\mathcal{M}}$  of states (i.e., first-order structures) closed under isomorphisms.
- A non-empty set  $\mathcal{I}_{\mathcal{M}} \subseteq \mathcal{S}_{\mathcal{M}}$  of initial states, also closed under isomorphisms.
- A set of extended-states  $\mathcal{E}_{\mathcal{M}} = \mathcal{S}_{\mathcal{M}} \times \mathcal{R}_{\mathcal{M}}$ .
- A transition function  $\tau_{\mathcal{M}}$  over  $\mathcal{E}_{\mathcal{M}}$  such that  $\tau_{\mathcal{M}}((S, R)) = (S, R) + \Delta(R, (S, R))$  for every  $(S, R) \in \mathcal{E}_{\mathcal{M}}$ , where  $R = \text{val}_S(\text{self})$  is the closed ASM rule in location *self* in the extended state  $(S, R)$ ,  $\Delta(R, (S, R))$  is the update set yielded by this rule in  $S$ , and  $(S, R) + \Delta(R, (S, R))$  denotes the extended-state obtained by applying to  $(R, s)$  the update set  $\Delta(R, (S, R))$ .

A run or computation of a reflective sequential ASM is a finite or infinite sequence of extended states  $(S_0, R_0), (S_1, R_1), (S_2, R_2), \dots$ , where  $S_0$  is a state in  $\mathcal{I}_{\mathcal{M}}$ ,  $R_0$  is the initial rule, and  $(S_{i+1}, R_{i+1}) = \tau_{\mathcal{M}}((S_i, R_i))$  holds for every  $i \geq 0$ .

Notice that for every  $R \in \mathcal{R}_{\mathcal{M}}$ , the functions in  $R$  allow us to examine and modify the “program” term stored in *self*. For instance, assume that the current value stored in *self* is the term  $\text{update}(f(t), s)$  and that we want to change it to  $\text{update}(f(t), s + 1)$ . Assuming the alphabet  $A$  includes the symbols “+” and “1”, the following sequential ASM rule updates *self* to the desired “program” term:  $\text{self} := \text{insertArgument}(\text{stringToTerm}(\text{TermToString}(\text{argumentNo}(\text{self}, 2)) \cdot + 1), 2, \text{self})$ .

Of course, it is quite cumbersome to update the rule in *self* by using the small set of background functions provided here. Nevertheless, this is enough to show that our approach works. In practice, we can use more convenient representations, for instance by means of complex values such as syntax trees, as well as more sophisticated functions to inspect and modify the ASM rules. Note that the kind of reflection that the RRM uses is a bit different to the one we propose in this work. We could call it “partial reflection”, since the sequence of actions performed in each transition, except for the queries to the relational store, never changes. We could then think of a different definition of the reflective ASM to represent partial reflection, where we only add to the sequential ASM a rule **eval**  $t$ , which takes a “program” term  $t$  as its argument, and interpret it as a sequential ASM rule (other than **eval**) which is then executed.

The next result shows the plausibility of our reflective ASM thesis.

**Theorem 1.** *Every reflective ASM  $\mathcal{M}$  is a RSA.*

*Proof (Sketch).* We need to show that  $\mathcal{M}$  satisfies the reflective sequential time, reflective abstract extended-state and reflective bounded exploration postulates. The first two postulates are already built into the definition of rASM, and the preservation of isomorphisms is straightforward.

In order to show that  $\mathcal{M}$  satisfies also the reflective bounded exploration postulate, we let  $W_{st} = \emptyset$  and  $W_{wt} = \{self\}$ . We see next that if two extended-states  $(S, R)$  and  $(S', R')$  of  $\mathcal{M}$  strongly coincide over  $W_{wt}$  then  $\Delta(R, (S, R)) = \Delta(R', (S', R'))$ . Since the states strongly coincide over  $W_{wt}$  we have that:

1.  $val_{(S,R)}(self) = val_{(S',R')}(self)$ .
2.  $val_{(S,R)}(raise_{(S,R)}(self)) = val_{(S',R')}(raise_{(S',R')}(self))$ .

Let  $W_r = \{r\}$  and  $W_{r'} = \{r'\}$ , where  $r$  and  $r'$  are the tuples of terms that result from the evaluation of *self* in  $(S, R)$  and  $(S', R')$ , respectively. From our definition of the “program” functions and the proof of the plausibility theorem of the sequential ASM thesis, we get that  $W_r$  and  $W_{r'}$  constitute, respectively, bounded exploration witnesses for the sequential ASM rules  $R$  and  $R'$ . In turn, by (1), we further have that  $W_r = W_{r'}$ . Finally, by (2) we get that  $(S, R)$  and  $(S', R')$  coincide on  $W_r = W_{r'}$ . Hence, by Gurevich’s bounded exploration postulate for sequential algorithms, we get that  $\Delta(R, (S, R)) = \Delta(R', (S', R'))$ . The plausibility theorem for RSA then follows.  $\square$

## 4 The Reflective Sequential ASM Thesis

We start by analysing an arbitrary RSA  $\mathcal{A}$ . Let  $W_{st} \cup W_{wt}$  be a bounded exploration witness for  $\mathcal{A}$  and let  $(S, P)$  be a state of  $\mathcal{A}$ . We define the set of *terms generated by  $W_{wt}$  in  $(S, P)$*  as follows:  $G_{W_{wt}}^{(S,P)} = \{raise_{(S,P)}(t) \mid t \in W_{wt}\}$ . We assume that  $W_{st} \cup G_{W_{wt}}^{(S,P)}$  is closed under sub-terms and call it the set of *critical terms of  $(S, P)$* .

The following lemma can be proven using the same argument as in the proof of the analogous Lemma 6.2 in the sequential ASM thesis [8].

**Lemma 1.** *If  $(f, (v_1, \dots, v_n), v_0)$  is an update in  $\Delta(\mathcal{A}, (S, P))$ , then  $v_0, v_1, \dots, v_n$  are values of critical terms of  $(S, P)$ .*

Lemma 1 implies that every update in  $\Delta(\mathcal{A}, (S, P))$  can be programmed by an update rule of the form  $f(t_1, \dots, t_n) := t_0$ , where the terms  $t_0, t_1, \dots, t_n$  are critical terms of  $(S, P)$ . To program the whole  $\Delta(\mathcal{A}, (S, P))$ , we define a sequential ASM rule  $r_{(S,P)}$  which is the parallel combination (by means of **par** rules) of all update rules in the following *finite* set:

$$\{f(t_1, \dots, t_n) := t_0 \mid t_0, t_1, \dots, t_n \in W_{st} \cup G_{W_{wt}}^{(S,P)} \text{ and} \\ (f, (val_{(S,P)}(t_1), \dots, val_{(S,P)}(t_n)), val_{(S,P)}(t_0)) \in \Delta(\mathcal{A}, (S, P))\}.$$

As  $W_{st} \cup G_{W_{wt}}^{(S,P)}$  is finite and the signature of  $(S, P)$  is also finite,  $r_{(S,P)}$  is well defined.

**Corollary 1.** *For every  $(S, P) \in \mathcal{S}_A$  there is a rule  $r_{(S,P)}$  such that:*

1.  $r_{(S,P)}$  uses only critical terms, i.e., terms in  $W_{st} \cup G_{W_{wt}}^{(S,P)}$ .
2.  $\Delta(r_{(S,P)}, (S, P)) = \Delta(\mathcal{A}, (S, P))$ .

From now on,  $r_{(S,P)}$  is as in the previous corollary.

**Lemma 2.** *If two extended-states  $(S, P)$  and  $(S', P')$  of  $\mathcal{A}$  strongly coincide over  $W_{st} \cup W_{wt}$ , then  $\Delta(r_{(S,P)}, (S', P')) = \Delta(\mathcal{A}, (S', P'))$ .*

*Proof.* As  $(S, P)$  and  $(S', P')$  strongly coincide over  $W_{st} \cup W_{wt}$ , we have that  $G_{W_{wt}}^{(S,P)} = G_{W_{wt}}^{(S',P')}$  and that, for every  $t \in W_{st} \cup G_{W_{wt}}^{(S,P)}$ ,  $val_{(S,P)}(t) = val_{(S',P')}(t)$ . As  $r_{(S,P)}$  only involves critical terms of  $(S, P)$ , i.e., terms in  $W_{st} \cup G_{W_{wt}}^{(S,P)}$ , we have that  $\Delta(r_{(S,P)}, (S, P)) = \Delta(r_{(S,P)}, (S', P'))$ . By Corollary 1,  $\Delta(r_{(S,P)}, (S, P)) = \Delta(\mathcal{A}, (S, P))$ . Finally, we obtain  $\Delta(\mathcal{A}, (S, P)) = \Delta(\mathcal{A}, (S', P'))$  by the reflective bounded exploration postulate.  $\square$

Let  $(S, P)$  and  $(S', P')$  be extended-states of  $\mathcal{A}$ . We say that  $(S', P')$  is *relative  $W[(S, P)]$ -equivalent* to  $(S, P)$  if  $G_{W_{wt}}^{(S',P')} = G_{W_{wt}}^{(S,P)}$ , and that they *coincide over  $W[(S, P)]$*  (in the sense of the sequential ASM thesis [8]) if  $val_{(S,P)}(t) = val_{(S',P')}(t)$  for all  $t \in W_{st} \cup G_{W_{wt}}^{(S,P)}$  (i.e., for all critical terms of  $(S, P)$ ).

The following is a straightforward corollary of Lemma 2 obtained by restricting the sets of updates to the locations in the “standard” sub-structure of the extended-states.  $\Delta_{st}$  denotes the subset of updates with function names which do *not* appear in  $\Sigma_{algo}$ .

**Corollary 2.** *If two extended-states  $(S, P)$  and  $(S', P')$  are relative  $W[(S, P)]$ -equivalent and coincide over  $W[(S, P)]$ , then we have  $\Delta_{st}(r_{(S,P)}, (S', P')) = \Delta_{st}(\mathcal{A}, (S', P'))$ .*

Consider the class  $\mathcal{C}[(S, P)]$  of relative  $W[(S, P)]$ -equivalent states of  $\mathcal{A}$ . Two states  $(S_1, P_1)$  and  $(S_2, P_2)$  of  $\mathcal{A}$  are  *$W$ -equivalent relative to  $\mathcal{C}[(S, P)]$*  iff  $(S_1, P_1), (S_2, P_2) \in \mathcal{C}[(S, P)]$  and  $E_{(S_1, P_1)} = E_{(S_2, P_2)}$ , where (for  $i = 1, 2$ )  $E_{(S_i, P_i)}(t_1, t_2) \equiv val_{(S_i, P_i)}(t_1) = val_{(S_i, P_i)}(t_2)$  is an equivalence relation in the set of critical terms of  $(S, P)$ .

**Lemma 3.** *If two extended-states  $(S_1, P_1)$  and  $(S_2, P_2)$  of  $\mathcal{A}$  are  $W$ -equivalent relative to  $\mathcal{C}[(S, P)]$ , then  $\Delta_{st}(r_{(S_1, P_1)}, (S_2, P_2)) = \Delta_{st}(\mathcal{A}, (S_2, P_2))$ .*

*Proof (sketch).* Note that if we assume  $\Delta_{st}(r_{(S_1, P_1)}, (S_3, P_3)) = \Delta_{st}(\mathcal{A}, (S_3, P_3))$  for a state  $(S_3, P_3) \in \mathcal{C}[(S, P)]$  with  $S_3$  isomorphic to  $S_2$ , then we get that  $\Delta_{st}(r_{(S_1, P_1)}, (S_2, P_2)) = \Delta_{st}(\mathcal{A}, (S_2, P_2))$ . This fact is analogous to Lemma 6.8 of the sequential ASM thesis [8] and can be proven in the same way. Thus, we just need to find an extended-state  $(S_3, P_3) \in \mathcal{C}[(S, P)]$  with  $S_3$  isomorphic to  $S_2$  and such that  $\Delta_{st}(r_{(S_1, P_1)}, (S_3, P_3)) = \Delta_{st}(\mathcal{A}, (S_3, P_3))$ .

Assume w.l.o.g. that the base sets of  $S_1$  and  $S_2$  are disjoint. Let  $S_3$  be the structure isomorphic to  $S_2$  which is obtained by replacing  $val_{S_2}(t)$  with  $val_{S_1}(t)$

for all critical terms  $t$  of  $(S, P)$ . This is well defined because  $(S_1, P_1)$  and  $(S_2, P_2)$  are  $W$ -equivalent relative to  $\mathcal{C}[(S, P)]$ . Take  $P_3 = P_2$ , then  $(S_3, P_3) \in \mathcal{C}[(S, P)]$ . By the reflective abstract state postulate,  $(S_3, P_3)$  is an extended-state of  $\mathcal{A}$ . Since  $(S_1, P_1)$  and  $(S_3, P_3)$  coincide over the set of critical terms of  $(S, P)$ , Corollary 2 gives  $\Delta_{st}(r_{(S_1, P_1)}, (S_3, P_3)) = \Delta_{st}(\mathcal{A}, (S_3, P_3))$ .  $\square$

Let  $\varphi_{(S, P)}$  be the following Boolean term:

$$\bigwedge_{\substack{t_i, t_j \in W_{st} \cup G_{W_{wt}}^{(S, P)} \\ \text{val}_{(S, P)}(t_i) = \text{val}_{(S, P)}(t_j)}} t_i = t_j \quad \wedge \quad \bigwedge_{\substack{t_i, t_j \in W_{st} \cup G_{W_{wt}}^{(S, P)} \\ \text{val}_{(S, P)}(t_i) \neq \text{val}_{(S, P)}(t_j)}} \neg(t_i = t_j).$$

As the set of critical terms of an extended-state  $(S, P)$  (i.e.,  $W_{st} \cup G_{W_{wt}}^{(S, P)}$ ) is finite, there is a finite set  $\{(S_1, P_1), \dots, (S_n, P_n)\}$  of states in  $\mathcal{C}[(S, P)]$  (the class of relative  $W[(S, P)]$ -equivalent states of  $\mathcal{A}$ ) such that every state in  $\mathcal{C}[(S, P)]$  is  $W$ -equivalent relative to  $\mathcal{C}[(S, P)]$  to one of the states  $(S_i, P_i)$ . Construct a rule **par if**  $\varphi_{(S_1, P_1)}$  **then**  $r_{(S_1, P_1)}$  **endif**  $\dots$  **if**  $\varphi_{(S_n, P_n)}$  **then**  $r_{(S_n, P_n)}$  **endif endpar**. Then the following result clearly follows from the previous lemmata.

**Lemma 4.**  $\Delta_{st}(r_{[(S, P)]}, (S_i, P_i)) = \Delta_{st}(\mathcal{A}, (S_i, P_i))$  for every extended-state  $(S_i, P_i) \in \mathcal{C}[(S, P)]$ , i.e., for every extended-state that is relative  $W[(S, P)]$ -equivalent to  $(S, P)$ .

Thus, for every class  $\mathcal{C}[(S_i, P_i)]$  of extended-states of  $\mathcal{A}$ , we have a corresponding rule  $r_{[(S_i, P_i)]}$  such that Lemma 4 holds. Now, we need to extend this result to all extended-states which belong to some run of  $\mathcal{A}$ , not just for the extended-states in the class  $\mathcal{C}[(S_i, P_i)]$ . Here is when the power of reflection becomes apparent.

Fix an arbitrary initial extended state  $(S, P)$  of  $\mathcal{A}$ . We define  $\mathcal{M}$  as the reflective ASM machine with  $\mathcal{E}_{\mathcal{M}} = \{(S_i, P'_i) \mid (S_i, P_i) \in \mathcal{E}_{\mathcal{A}} \text{ and } P'_i \text{ is the "self" representation of } r_{[(S_i, P_i)]}\}$  and  $\mathcal{I}_{\mathcal{M}} = \{(S_i, P') \mid S_i \in \mathcal{I}_{\mathcal{A}} \text{ and } P' \text{ is the "self" representation of } r_{[(S, P)]}\}$ .

**Lemma 5.** For every run of  $\mathcal{A}$  of the form  $(S_0, P_0), (S_1, P_1), \dots$  and corresponding run of  $\mathcal{M}$  of the form  $(S'_0, P'_0), (S'_1, P'_1), \dots$  with  $S_0 = S'_0$ , it holds that  $\Delta_{st}(r_{[(S_i, P'_i)]}, (S'_i, P'_i)) = \Delta_{st}(\mathcal{A}, (S_i, P_i))$ .

*Proof (Sketch).* We prove it by induction on an arbitrary run of  $\mathcal{A}$ . By the reflective sequential time postulate, we know that every initial extended-state  $(S_0, P_0)$  of every run of  $\mathcal{A}$  is relative  $W[(S, P)]$ -equivalent to the initial extended-state  $(S, P)$  used in the construction of  $\mathcal{M}$ . Thus, we get from Lemma 4 that  $\Delta_{st}(r_{[(S_0, P'_0)]}, (S_0, P_0)) = \Delta_{st}(\mathcal{A}, (S_0, P_0))$ . Given the restriction to “standard” updates which do not involve updates to the algorithm, we have

$$\Delta_{st}(r_{[(S_0, P'_0)]}, (S'_0, P'_0)) = \Delta_{st}(\mathcal{A}, (S_0, P_0)).$$

Regarding the inductive step. As  $P_i$  is a sequential algorithm, it is captured by a sequential ASM  $M_i$ . Moreover, due to Gurevich's proof of the sequential ASM thesis [8], the rule has the form

**par if  $\psi_1$  then  $r_1$  endif ... if  $\psi_k$  then  $r_k$  endif endpar,**

where each  $r_j$  is a **par** block of assignment rules. All  $\psi_j$  and  $r_j$  involve critical terms defined by a bounded exploration witness of  $P_i$  such as  $W_{st} \cup G_{W_{wt}}^{(S_i, P_i)}$ .

Due to construction of  $r_{[(S'_i, P'_i)]}$ , we have that  $W_{st} \cup G_{W_{wt}}^{(S_i, P_i)}$  is bounded exploration witness of the “self” representation  $P'_i$  of  $r_{[(S'_i, P'_i)]}$ . In turn, by construction of  $\mathcal{M}$  it can be shown that  $W_{st} \cup W_{wt}$  is a bounded exploration witness for  $\mathcal{M}$ . Thus, the updates in  $\Delta_{st}(r_{[(S'_i, P'_i)]}, (S'_i, P'_i))$ , transform the “self” representation  $P'_i$  of  $r_{[(S'_i, P'_i)]}$  into the “self” representation  $P'_{i+1}$  of  $r_{[(S'_i, P'_{i+1})]}$ . Since from the inductive hypothesis it can be shown that  $S'_i = S_i$ , we get that  $\Delta_{st}(r_{[(S'_{i+1}, P'_{i+1})]}, (S'_{i+1}, P'_{i+1})) = \Delta_{st}(\mathcal{A}, (S_{i+1}, P_{i+1}))$ .  $\square$

Using the previous key lemma, it is not difficult to show that every run of  $\mathcal{A}$  of the form  $(S_0, P_0), (S_1, P_1), \dots$  is *essentially equivalent* to the corresponding run of  $\mathcal{M}$  of the form  $(S'_0, P'_0), (S'_1, P'_1), \dots$  with  $S_0 = S'_0$ , i.e., that  $S_i = S'_i$  and that the restriction of  $P_i$  and  $P'_i$  to the signature  $\Sigma$  of  $S_i$  and  $S'_i$  results in non-reflective algorithms which are behavioural equivalent. This implies our main result.

**Theorem 2.** *For every RSA  $\mathcal{A}$  there is a behavioural equivalent rASM machine  $\mathcal{M}$ .*

## 5 Conclusion

In this paper we investigated a behavioural theory for reflective sequential algorithms (RSAs) following our conjecture in [7]. Grounded in related work concerning behavioural theories for sequential algorithms [8], (synchronous) parallel algorithms [6], non-deterministic algorithms [10] and concurrent algorithms [4] we developed a set of abstract postulates characterising RSAs, extended ASMs to reflective Abstract State Machines (rASMs), and formally sketched the proof that any RSA as stipulated by the postulates can be step-by-step simulated by a rASM. The key contributions are the postulates themselves, as they provide a language-independent definition of RSAs and the characterisation proof.

With this behavioural theory we lay the foundations for rigorous development of reflective algorithms and thus self-adaptive systems. However, several open tasks still have to be addressed before a general behavioural theory of *evolving concurrent systems* (ECS) will be reached. It is required to combine the behavioural theory developed in this paper with those for parallel algorithms thus proving a behavioural theory for reflective parallel algorithms, and with the theory of concurrency thus proving a behavioural theory for concurrent reflective systems, i.e. ECS. In view of the similarity of arguments in the separate behavioural theses this integration appears plausible, but nonetheless constitutes

a mathematically challenging problem. Furthermore, for rigorous development extensions to the refinement method for ASMs [3] and to the logic used for verification [14] will be necessary. These will be addressed in follow-on research.

## References

1. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. *ACM Trans. Comput. Logic* **4**(4), 578–651 (2003)
2. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms: correction and extension. *ACM Trans. Comp. Logic* **9**(3), 19 (2008)
3. Börger, E.: The ASM refinement method. *Formal Aspects Comput.* **15**, 237–257 (2003)
4. Börger, E., Schewe, K.D.: Concurrent abstract state machines. *Acta Inform.* **53**(5), 469–492 (2016)
5. Börger, E., Stärk, R.: *Abstract State Machines*. Springer, Heidelberg (2003). <https://doi.org/10.1007/978-3-642-18216-7>
6. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing - simplified parallel ASM thesis. *Theor. Comput. Sci.* **649**, 25–53 (2016)
7. Ferrarotti, F., Tec, L., Torres, J.M.T.: Towards an ASM thesis for reflective sequential algorithms. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) *ABZ 2016*. LNCS, vol. 9675, pp. 244–249. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-33600-8\\_16](https://doi.org/10.1007/978-3-319-33600-8_16)
8. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic* **1**(1), 77–111 (2000)
9. Riccobene, E., Scandurra, P.: Towards ASM-based formal specification of self-adaptive systems. In: Ameer, Y.A., Schewe, K.D. (eds.) *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. LNCS, vol. 8477, pp. 204–209. Springer, Heidelberg (2014). <https://doi.org/10.1007/978-3-662-43652-3>
10. Schewe, K.D., Wang, Q.: A customised ASM thesis for database transformations. *Acta Cybern.* **19**(4), 765–805 (2010)
11. Smith, B.C.: Reflection and semantics in LISP. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1984*, pp. 23–35. ACM (1984)
12. Stemple, D., et al.: Type-safe linguistic reflection: a generator technology. In: Atkinson, M., Welland, R. (eds.) *Fully Integrated Data Environments*. Esprit Basic Research Series, pp. 158–188. Springer, Heidelberg (2000). [https://doi.org/10.1007/978-3-642-59623-0\\_8](https://doi.org/10.1007/978-3-642-59623-0_8)
13. Van den Bussche, J., Van Gucht, D., Vossen, G.: Reflective programming in the relational algebra. *J. Comput. Syst. Sci.* **52**(3), 537–549 (1996)
14. Wang, Q., Ferrarotti, F., Schewe, K.D., Tec, L.: A complete logic for non-deterministic database transformations. *CoRR* abs/1602.07486 (2016). <http://arxiv.org/abs/1602.07486>

# Lightweight Non-intrusive Virtual Machine Introspection

Natalia Fursova<sup>1,2</sup>, Pavel Dovgalyuk<sup>1,2</sup>(✉), Ivan Vasiliev<sup>1,2</sup>,  
and Vladimir Makarov<sup>1,2</sup>

<sup>1</sup> Novgorod State University, Velikiy Novgorod, Russia

<sup>2</sup> Institute for System Programming of the Russian Academy of Sciences,  
Moscow, Russia

{natalia.fursova,pavel.dovgaluk,ivan.vasiliev,  
vladimir.makarov}@ispras.ru

**Abstract.** Dynamic analysis is an important technology for different phases of the software life cycle. Dynamic analysis is used for profiling, malware analysis, intrusion detection, protocol reverse engineering, software testing, and many other activities. This paper presents a lightweight approach for monitoring of systems using virtual machines. Our approach is based on non-intrusive virtual machine introspection, which provides system-wide analysis capabilities. We reuse ABI of the platform to be analyzed for creating introspection tools. We show how to recover the part of kernel-level information related to the system calls executed on the guest machine. The paper describes how to use this approach to create plugin-based analysis framework for simulator QEMU and evaluates performance overhead for these plugins.

## 1 Introduction

Non-intrusive dynamic analysis is an important technology for different phases of the software life cycle. Dynamic analysis is used for profiling, malware analysis, intrusion detection, protocol reverse engineering, software testing, and many other activities [1, 6, 12, 14, 20].

System-wide analysis has several advantages. First, it provides full system view. One can perform analysis of kernel and user code, including drivers and all executed processes. Second, analysis of the code executed in simulator is non-intrusive. One can run code in simulated environment without exposing analysis software to this code. Analyzer can also inspect BIOS and startup code of the guest OS. Non-intrusive analysis may be also used when it is not possible to load any new code into the guest system, e.g., when execution of the system is replayed [9, 10].

Full-system live analysis is usually performed by analysis of the binary code and requires recovering of guest kernel- and user-level data structures. Semantic gap between binary representation and high-level data structures hampers the analysis [4]. Virtual machine introspection is the most used approach to bridge

the semantic gap between binary and source representations of data structures while examining the whole system execution [20]. Introspection tool examines the executed guest machine and recovers significant information about executed processes.

There are several approaches to virtual machine introspection (VMI). State-of-the-art approaches reuse source code of kernels [14, 15] or binary code of user-level agent, installed into the guest system [16, 22]. First approach makes analysis tightly coupled with internal OS structure, and second one is intrusive and changes the behavior of the system.

We present an introspection approach which reuses application binary interface (ABI). As well as reusing sources, reusing ABI exploits the knowledge about kernels, but in our case the number of entities to be analyzed is significantly smaller. We also track CPU-level events like TLB miss or interrupt requests to recover execution context switching and parse executables to monitor API function calls.

Our work is based on the open source multi-platform simulator QEMU [2]. QEMU is capable of running virtual machines based on commodity platforms (e.g., i386, ARM, MIPS, PowerPC). We added new plugin subsystem and subsystem for dynamic instrumentation into QEMU. We also created several plugins for monitoring system calls, file operations, API functions, and process operations.

In summary, this paper makes the following contributions:

1. An approach to virtual machine introspection. We reuse platform ABI to extract kernel- and process-level information, as described in Sect. 2. Reusing ABI reduces maintainance efforts required to support the analysis plugins and apply them to other versions of guest operating systems.
2. Non-intrusive approach to system-wide system call monitoring, which is described in Sect. 2.1. We instrument guest code executed in QEMU to insert monitoring functions. Non-intrusive monitoring may be used with execution replay, when guest agent could not be loaded into the virtual machine. Non-intrusive approach is also suitable for firmwares that cannot be modified directly.
3. Layered model of plugins framework. Introspection and analysis plugins in our model may be divided into logical layers that simplify their communications between them, as described in Sect. 3. Plugins raise abstraction level of recovered VM information for reconstructing OS-level and application-level data structures.
4. Instrumentation layer and plugin support for open source simulator QEMU. Plugins use instrumentation to insert callbacks into the translated guest code. We created thin instrumentation layer which allows easy migration of the plugins to newer QEMU versions.
5. Plugins for monitoring file and process operations for Linux and Windows guest OSes. We created several plugins that analyse system calls in platform-dependent way. These plugins transform platform-dependent system call information into platform-independent format. Higher-level plugins reuse this platform-independent information to perform monitoring or other activities. We evaluate the performance of our approach in Sect. 4.



## 2 Reusing ABI for Virtual Machine Introspection

One can monitor target application activities with standalone analysis applications running on the same system. However, using virtual machine for monitoring and analysis has several advantages over single process monitoring:

- Analysis of kernel and drivers' code. Introspection of virtual machine can trace data paths through whole virtual machine including kernel code.
- Simultaneous analysis of all executed processes. One can analyze interactions between processes or monitor any of the newly started processes without reconfiguring the analyzer.
- Non-intrusive analysis. Software executed in virtual machine cannot notice that it is analyzed when analysis code is executed outside the virtual machine. Analysis also does not affect timings (in case of using virtual guest clock) and lies outside the guest memory.
- OS-independence. As we show in Sect. 3, introspection engine may be designed as OS-independent component. One can execute any operating system and gain analysis possibilities for them. Of course, this OS should use the same ABI we designed introspection for.
- Platform-independence for high-level analysis algorithms. One can design analysis algorithm in platform-independent way. Analysis code can use our platform-dependent plugins that provide data, recovered from hardware and OS level, in platform-independent format. Details of this mechanism is described in Sect. 3.

On the other hand, whole virtual machine introspection incurs additional overhead, compared to native execution, because of virtualization. Introspection overhead may depend on executed applications. We evaluate performance of our method and compare it with single-process tracer in Sect. 4.

Guest machine will not detect this overhead when virtual time is calculated as a number of executed guest instructions multiplied by some constant. In this case overhead may be noticed only by observing virtual machine's environment (e.g., sending network packet and checking the response time).

Whole virtual machine monitoring also have to duplicate several services that application-level analysis program may reuse from operating system. One of these is monitoring execution context switching, because analysis algorithm needs to know which program currently executes on virtual machine.

There are two main introspection approaches. The first one is reusing sources. It is based on knowledge about data structures and recovering them from the guest memory [14, 15]. This approach requires adjusting analysis algorithms for every new build of the target operating system. E.g., there are many different builds of Linux kernel that differ by core patches, compiler versions, and compilation directives. Analysis module has to adjust offsets of the data structures in memory to recover them from the memory dump.

Another approach is executing guest monitoring applications to obtain requested data. With this approach guest agent can be installed into the guest

system to monitor its OS and applications [16,22]. This method is intrusive—it can change the behavior of the guest system.

We propose a new introspection approach, which is driven by two main requirements: no modification of guest OS and applications, and making an OS-specific part of the system as small as possible. The first requirement is satisfied by using the virtual machine and inspecting its CPU and memory state. The second requirement is satisfied by using small ABI-specific part of the code to transform kernel data structures into platform-independent representation. All analysis and monitoring algorithms can work with this abstract representation only.

Application binary interface includes the list of system calls, functions calling convention, data alignment, execution files format, stack format, and registers usage pattern. ABIs are designed for the hardware or software platforms and remains mostly unchanged with platforms evolution for the sake of backward compatibility.

One of the significant parts of ABI is system call interface. System call functions in Linux are identified by integer passed as a parameter in one of the registers. These identifiers never change. Therefore maintenance efforts for Linux introspection modules in our approach will include only adding new system calls (if needed) and supporting arguments passing agreement for the new platforms added to the analysis scope.

Due to its (almost) static nature we can reuse ABI to perform introspection for a wide range of platforms. E.g., parameters of kernel data structures (fields, offsets, alignment) have greater volatility than ABI, because kernel structures binary representation depends on used compiler, build options, and kernel code version. Complete information about kernel structures may be unavailable (e.g., in case of Windows), in contrast to system call identifiers and parameters.

Our technique does not depend on internal kernel data structures, because we build OS-agnostic representation of system internals. Analyzed code should work on a machine with a trusted kernel, because we rely on correctness of the system calls.

## 2.1 System Call Monitoring

The key to recovering information about executed processes in our approach is a system call monitoring. We hook system calls to capture OS-specific information from the virtual machine.

System call monitoring approach is based on reusing platform ABI. It allows hooking file, thread and process, memory mapping operations, and many other. System call interface is well documented and does not change with every version of the operating system. We hook system calls and parse their parameters and return values instead of monitoring guest memory to find kernel data structures.

There are few possibilities for calling system functions for each platform. System call is usually performed by specific instruction: `syscall`, `sysenter`, `int 0x80`, `int 0x2e` for x86/64, `svc #0` for ARM, `syscall` for MIPS, `sc` for PowerPC. These instructions jump to the kernel code which executes the

requested function. Concrete instruction used for system call may be OS-dependent. E.g., Windows NT uses `int 0x2e`, and Windows XP uses `sysenter`.

## 2.2 Execution Context

To match system-call and system-return instructions we save execution context when call is about to execute. When execution reaches return instruction, we find the context, recover system call parameters for it, and query function's return value from the guest memory or registers.

In our case execution context is the current process id and stack pointer. We identify processes by page directory address (e.g., `CR3` for x86 or `CP15.c2` for ARM) [14]. Context information is updated every time this register is reloaded.

## 3 Introspection Plugins for QEMU

With multi-platform support one can make analysis tools that can be executed on many platforms. Therefore we chose simulator QEMU for implementation of our monitoring method. QEMU is a multi-target simulator supporting commodity hardware platforms like x86, x86-64, ARM, MIPS, and PowerPC [2]. It translates guest binary code into host binary code and then executes it. Due to binary translation and caching of the translated code, QEMU works faster than interpreters, e.g., Bosch [17].

QEMU does not provide instrumentation interface for researchers and developers. Adding any instrumentation extension becomes a challenge for the developer, because QEMU internals are poor documented. Therefore, providing interface for creating plugins reduces development overhead for instrumentation functions.

We implemented virtual machine introspection as a set of plugins for the simulator. Several plugins depend on ABI or hardware and contain platform-dependent code which recovers system-level events and data. Other plugins use these events and structures to maintain abstract representation of kernel objects. Abstract kernel objects are OS-agnostic and can be used within any platform.

We divided all projected analysis tasks between plugins. Altogether they form analysis framework for recovering and logging data structures and system events.

Plugins in our framework can be divided between several layers of abstraction, as shown in Fig. 1. These layers represent information that plugins recover from the execution of the binary code. Plugin layers do not represent structure of an operating system. Layers correspond to the order of recovering OS-level and user-level information from the executed binary code.

Plugins on different layers interact by exchanging messages. Messages correspond to particular events that denote simulator's external communications or changes in virtual machine state.

### 3.1 Simulator Events for Plugins

Hardware layer includes events and data generated by the simulator. The following list describes the events tracked by the simulator. Simulator notifies the plugins when one of these events happens.

1. **Translation events.** Before QEMU executes any instruction, it has to translate guest code to host binary code through the target-independent internal representation [2].
2. **Execution events.** These events denote block or instruction execution.
3. **Memory events.** These events allow creating memory monitoring tools. TLB misses may also be used for detection of creating new execution context [14].
4. **Hardware events.** Notifications for virtual hardware events may be used to monitor specific virtual hardware operations. One can debug new virtual devices or software drivers for the existing devices using these notifications.
5. **Input events.** Notifications of input events may be used for user operations monitoring.
6. **System events.** This set of events include events related to the whole system, e.g., machine reset.

Plugins process hardware-level events and recover operating system internals step-by-step, raising the level of abstraction. When the required abstractions are recovered, higher-level plugins can monitor their state and behavior.

Next to hardware layer is the layer for monitoring files, threads, processes, and other OS objects. Files accesses are monitored by hooking corresponding system calls that were recovered in the previous layer. Process creation operations are monitored in the same manner.

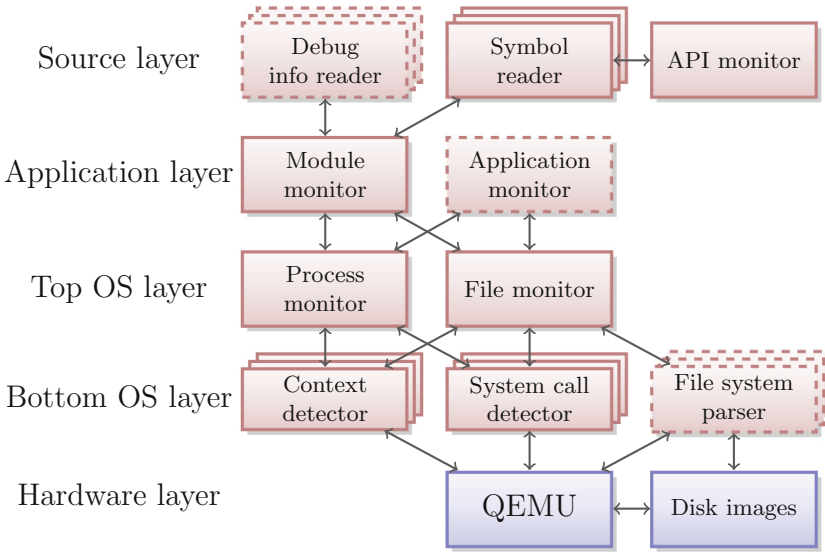
Application layer includes plugins that recover modules, applications, and other user-level objects. Plugins at source layer extract symbolic and debugging information from the executables to provide user-friendly information about debugged or analyzed application.

### 3.2 Guest Code Instrumentation

QEMU uses dynamic translation to execute the guest code. Every guest instruction is transformed into a sequence of host instructions that simulate behavior of the guest one. QEMU joins translated instructions into blocks. Translation block is a continuous sequence of instructions. It means that execution of the block always starts at the first instruction and ends at the last one.

To invoke our own code on system call execution we embed callbacks into translated code, as shown in Fig. 2. These callbacks recover arguments of the system calls and their return values. Parameters are recovered on system call entry and return values on system call exit instruction.

When system call code is translated, system call plugin embeds two callbacks into the generated code. The first callback is required to check that called



**Fig. 1.** Plugins for virtual machine introspection. Dashed modules are not implemented yet.

function is related to tracked operations. This callback also queries system call arguments from the guest memory. The second callback is invoked when execution of the system call is finished. This callback is embedded before return instruction execution. It reads return value from the CPU state and sends a message with description of the file operation to all listeners.

### 3.3 File Monitoring

In this section we describe implementation of file monitoring plugins. Different operating systems have different sets of system functions. Therefore, we had to create one system call detection plugin for every of the OS families. We also created file monitor plugin. This plugin is independent from executing guest OS and guest hardware architecture. It operates with system call and file abstractions.

Main system calls for handling file operations are `NtCreateFile`, `NtOpenFile`, `NtReadFile`, `NtWriteFile`, `NtClose` in Windows, and `creat`, `open`, `read`, `write`, `close` in Linux.

However, hooking system calls is not enough for complete file monitoring, because of hard- and soft-links in the file system. Named file even may not be written on disk at all. We will need disk image analysis to monitor such situations. As a first step to full-stack disk state analysis we implemented file monitoring plugin, which hooks file operation system calls.

File monitoring plugin maintains the list of the open files to match `close` system calls with the files, because close operations may be used to free other system resources (`NtClose` is used to close all handles in Windows).

## Translation block to be executed

```

0xb7707010: mov    %ebx,%edx
0xb7707012: mov    0x8(%esp),%ecx
0xb7707016: mov    0x4(%esp),%ebx
0xb770701a: mov    $0x21,%eax
0xb770701f: int   $0x80

```

## Intermediate representation of the translation block

```

ld_i32 tmp11,env,$0xfffffffffff8
movi_i32 tmp12,$0x0
brcond_i32 tmp11,tmp12,ne,$L0
---- b7707010 00000000
mov_i32 tmp0,ebx
mov_i32 edx,tmp0
---- b7707012 00000000
movi_i32 tmp11,$0x8
add_i32 tmp2,esp,tmp11
qemu_ld_i32 tmp0,tmp2,leul,1
mov_i32 ecx,tmp0
---- b7707016 00000000
movi_i32 tmp11,$0x4
add_i32 tmp2,esp,tmp11
qemu_ld_i32 tmp0,tmp2,leul,1
mov_i32 ebx,tmp0
---- b770701a 00000000
movi_i32 tmp0,$0x21
mov_i32 eax,tmp0
---- b770701f 00000000
+ movi_i64 tmp13,$0xb7707020
+ movi_i64 tmp14,$0x7fef9a788670
+ call start_system_call,$0x0,$0,tmp13,tmp14
movi_i32 tmp3,$0xfffffffffb770701f
st_i32 tmp3,env,$0x20
movi_i32 tmp11,$0x2
movi_i32 tmp12,$0x80
call raise_interrupt,$0x0,$0,env,tmp12,tmp11
set_label $L0
exit_tb $0x7fef8e6dca13

```

**Fig. 2.** Translation block with int 0x80 instrumentation callback. Instructions marked with ‘+’ are inserted by instrumentation.

File monitoring is performed by starting one of the platform-specific system call detectors and file monitor plugin. System call plugin watches the executed system functions and filters calls related to the file operations. Information about file operations are passed to the file monitoring plugin, which matches open and close operations and logs file accesses.

### 3.4 Mapping Files to the Memory

Mapping of the files to the memory (or IO mapping) is a widely used technique in the commodity operating systems. With IO mapping files are not read through read or write functions.

Files are mapped with `mmap` function in Linux and with pair of `NtCreateSection` and `NtMapViewOfSection` functions in Windows. We hook them to detect whether specific file was mapped to the memory or not.

Operating systems uses memory-mapped files to load executables and dynamic libraries. Benefits of mapping executables are lazy loading, when non-accessed parts of a file are not read from the disk, and sharing between processes—OS may load a file once and map it to the virtual address space of multiple processes.

We hook mapping/unmapping operations in file monitoring plugin. This information is used to detect executable image loading and parse those images to extract API functions addresses.

### 3.5 API Monitoring

Monitoring of API calls may be useful in itself (e.g., for detecting malware), and also for recovering more system information, than from the system calls.

First of all, we implemented monitoring of API functions for Windows, because we needed information returned by `CreateProcess` function of `kernel32.dll` dynamic library to recover the list of the executed processes in process monitoring plugin.

Dynamically linked modules basically have one of the two image formats: ELF or PE. Formats of the executable images are well documented. Therefore we may use them to extract information of the API functions offsets.

How do we detect which module is executed? We monitor mapping function calls, as described in Sect. 3.4. In Windows files with system dynamic libraries are open once and then mapped with `NtOpenSection` and `NtMapViewOfSection` functions.

We instrument the entry points of the API functions located in the loaded library. When the code at that addresses is about to translate, we insert monitoring function call, which reports to the user that API function is called, and reads function's parameters from the memory to pass it to the other introspection plugins, if it is needed.

### 3.6 Process Monitoring

Process monitoring plugin provides the list of the currently executed guest processes to the user. To do this it monitors process creation and destruction events. For each discovered process it stores the following tuple:

- Execution context. We use page directory base register (e.g., `CR3` register for x86) to identify the execution context. Every process in modern operating systems has its own virtual address space. Physical page mapping is described with page directory, which is unique for every process.
- Process id assigned by operating system. Process id is not needed for further behavior analysis, but it is required for user's convenience. E.g., user can compare output of `ps` utility with information gathered with introspection plugins, when it is possible.
- Name of the executable image. This name is recovered from the file open system call, when executable is about to be mapped or loaded to the memory.

There are many differences between system calls parameters and execution sequence in different operating systems. E.g., Linux uses `fork` and `clone` functions for creation of the processes, and `execve` for running the programs. Windows uses `NtCreateProcess` for creating process without any threads and `NtCreateThread` for adding new threads. We use parameters and return values of these functions to reconstruct the list of the running processes.

Technique for capturing process creation and destruction system calls is the same as for file monitoring. We instrument system call and system call return instructions and analyze called function and its parameters.

However, `NtCreateProcess` does not expose enough information. We cannot directly retrieve id of the created process. Therefore we also hook `CreateProcess` API function. It returns process id after creating new process. With this function we match execution context and image file with process id. This information is used to output the list of the currently executed processes.

## 4 Evaluation

After implementing and testing file monitoring plugins with Windows and Linux guests we measured its performance overhead. We executed QEMU on a machine with Intel Core i7 CPU with 8 cores at 3.40 GHz, 8 GB RAM, 500 GB HDD, and 64-bit Ubuntu 14.04. Virtual machine on i386 platform had 128 MB of memory. We used Windows XP and Arch Linux as guest OSes.

We ran several tests: booting Windows XP, booting Linux, downloading 255Mb file under Linux, packing downloaded file with gzip, and unpacking the created archive. At first, we measured system call instrumentation overhead by enabling instrumentation of system calls and filtering the file operations without issuing any notifications. Second group of tests included measuring the file operations logging overhead by enabling file monitoring plugin. File logging plugin writes information about all file operations in the system. It also prints contents of read and write buffers to the log. All measurement results are presented in Table 1.

**Table 1.** Instrumentation and introspection performance

	Simulation	Instrumentation		File logging	
	Time, sec	Time, sec	Overhead, %	Time, sec	Overhead, %
Loading Windows	63	65	3.2	69	9.5
Loading Arch Linux	32	33	3.1	35	9.3
Downloading 255 MB file	30	32	6.7	101	237
Packing 255 MB file	106	113	6.6	284	168
Unpacking archive	18	19	5.6	28	56

In most cases instrumentation of system calls incurs very low overhead (5.1% on average). Such a small overhead allows using our approach for online monitoring and offline analysis. Logging of download/pack/unpack incurs greater



overhead, than booting OS, because file log includes contents of buffers for all read/write operations.

We also compared performance overhead of our system-wide instrumentation with application-level `strace` system call monitor utility. We ran latter two tests (packing and unpacking) with enabled system call tracing. Overhead of `strace` is presented in Table 2. We do not take virtualization overhead of our method into account, because our approach and `strace` have different application scopes. When non-intrusive introspection is a requirement, using virtual machines is one of the best options. If `strace` will be used under virtual machine, it will show similar relative overhead. Therefore our instrumentation of all executing processes looks competitive with instrumentation of a single process with `strace`.

**Table 2.** `strace` execution overhead

	Execution	Tracing	
	Time, sec	Time, sec	Overhead, %
Packing 255 MB file	6.0	8.0	33
Unpacking archive	1.6	5.0	213

Paper with description of DECAF dynamic analysis and instrumentation framework presents measurements of the overhead for OS booting [14]. According to this paper, DECAF VMI incurs 22% overhead for booting Windows XP on x86. TCG plugins framework provided only simple plugins for instruction counting, therefore it cannot be directly compared with our approach. TCG plugins framework incurs 3%–3000% overhead for this tracing task, depending on performed actions and printed debug data [13]. E.g., simplest instruction counting plugin incurs 3% overhead, calling empty function in every translation block—25%. QTrace instrumentation framework incurs 90% overhead for instruction counting plugin [21]. According to this data, our tool is competitive with other system-wide instrumentation and monitoring frameworks.

## 5 Related Work

In this section we give a revision of previous studies carried out by other researchers.

RTKDSM system leverages the rich OS analysis capabilities of Volatility, an open source computer forensics framework, to significantly simplify and automate analysis of VM execution states [15]. RTKDSM system performs real-time monitoring of the operating system state in the virtual machine. Volatility is intended to analyze memory dumps. RTKDSM system provides guest memory directly from virtual machine instead of memory dump. This approach is pretty straightforward. But dump-targeted Volatility architecture is not effective in

monitoring mode, because it has to analyse the whole dump every time when information is needed. Therefore, RTKDSM system uses Volatility only for locating the OS-specific data structures. RTKDSM system uses its own monitoring agent to keep track of the changes in these structures. The main limitation of RTKDSM system is targeting to x86 platform, because of using Xen hypervisor for the virtual machine.

PinOS is the framework for whole-system dynamic instrumentation [3]. It is an extension of Pin—application-level dynamic instrumentation framework [19]. PinOS is built on top of Xen virtual machine monitor. PinOS uses the same dynamic translation technique as Pin. Therefore, PinOS can use the plugins already developed for Pin. PinOS is limited by Xen and Pin, therefore it can only boot Linux on x86 virtual machine. PinOS incurs significant execution slowdown (up to 120x) even without any instrumentation. Its source code was not published. That’s why we couldn’t download and evaluate it.

QTrace is an instrumentation extension API developed on top of QEMU [21]. It is targeted to full-system analysis and analysis of unmonified applications supplied in binary form. The paper presents validation of QTrace instrumentation—performance profiling results for SPEC2006 are compared with Pin [19] and BOSCH [17]. QTrace incurs 90% overhead for instruction counting plugin. Proof-of-concept version is available on github<sup>1</sup>, but it does not provide any features except register and memory access tracing.

DECAF is a platform-neutral whole-system binary dynamic analysis framework [14]. DECAF is built upon QEMU and provides event-driven programming interface for custom analysis plugins. It reconstructs OS-level semantic view with virtual machine introspection engine. DECAF addresses the question “when to reconstruct” by monitoring hardware-level events. Events can be monitored for specific process or kernel module. DECAF supports Windows and Linux operating systems, ARM and x86 hardware platforms. The main drawbacks of DECAF is 15% performance overhead without any instrumentation and using old version of QEMU.

Dolan-Gavitt et al. describes technique of mining memory accesses for introspection [5–7]. They created a prototype system, Tappan Zee Bridge (TZB), which is focused on recovering user-level information. TZB observes memory accesses in runtime to find points in the program that access security-relevant information. This system cannot work online, because it incurs significant overhead.

They also created dynamic analysis framework named PANDA, which allows creating instrumenting tools. There is a bunch of introspection plugins for PANDA: system call monitoring in an emulated x86 or ARM guest, and call-stack monitoring. However, these introspection plugins do not extract higher-level information than just system call sequence.

Virtuoso analyses dynamic traces of small in-guest introspection programs to produce similar programs that work on a host machine [8]. But execution traces may be incomplete, making generated introspection programs behave incorrectly.

<sup>1</sup> <https://github.com/x-y-z/QTRACE>.

VMST tool demonstrates process out-grafting [11]. It analyzes data flow of the program in the trusted system and redirects data memory accesses from trusted system to the untrusted one. However, VMST supports limited subset of system calls, because it redirects guest memory accesses only. E.g., system calls for file operations cannot be redirected. VMST cannot be used for online monitoring, because of its slowdown of 9.3X on average.

Lefebvre et al. presents an approach to execution mining [18]. This approach is targeted to understanding operating systems behavior. Tralfamadore system described in that paper records detailed CPU-level traces of virtual machine execution. Execution traces may be analyzed with a library of different operators allowing to create new dynamic analyses. Operators include trace parsing, context identification (threads, interrupts), functions invocations detection, memory allocations detection, memory accesses tracer, and heap objects tracer. However, Tralfamadore cannot work online. It analyses only previously recorded traces.

## 6 Conclusion

In this work we described a promising approach for virtual machine introspection through monitoring of the system calls and virtual hardware events. Narrowing the semantic gap with virtual machine introspection will allow creating analysis, monitoring, and debugging tools. System call detector plugin for virtual machine is the basis of our framework. It analyses the executed instructions and filters system calls from the control flow. We showed that this approach is practical by creating the plugins for API functions, file and process operations monitoring. These plugins may be used for non-intrusive logging of Windows- and Linux-based guest systems. One can use it for debugging of user or kernel code, malware analysis, and other runtime validation tasks. These plugins are the proof-of-concept and a basis of modular dynamic analysis framework. Full framework will provide introspection plugins for dynamic analysis of kernels and applications.

**Acknowledgment.** The work was partially supported by the Ministry of Education and Science of Russia, research project No. 2.6146.2017/8.9.

## References

1. Azmandian, F., Moffie, M., Alshawabkeh, M., Dy, J., Aslam, J., Kaeli, D.: Virtual machine monitor-based lightweight intrusion detection. *SIGOPS Oper. Syst. Rev.* **45**(2), 38–53 (2011). <http://doi.acm.org/10.1145/2007183.2007189>
2. Bellard, F.: Qemu, a fast and portable dynamic translator. In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC 2005*, p. 41. USENIX Association, Berkeley (2005). <http://dl.acm.org/citation.cfm?id=1247360.1247401>
3. Bungale, P.P., Luk, C.K.: Pinos: a programmable framework for whole-system dynamic instrumentation. In: *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007*, pp. 137–147. ACM, New York (2007). <http://doi.acm.org/10.1145/1254810.1254830>

4. Chen, P., Noble, B.: When virtual is better than real [operating system relocation to virtual machines]. In: Proceedings of the Eighth Workshop on Hot Topics in Operating Systems, 2001, pp. 133–138, May 2001
5. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable reverse engineering for the greater good with panda, October 2014
6. Dolan-Gavitt, B., Hodosh, J., Hulin, P., Leek, T., Whelan, R.: Repeatable reverse engineering with panda. In: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW-5, pp. 4:1–4:11. ACM, New York (2015). <http://doi.acm.org/10.1145/2843859.2843867>
7. Dolan-Gavitt, B., Leek, T., Hodosh, J., Lee, W.: Tappan zee (north) bridge: mining memory accesses for introspection. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & #38; Communications Security, CCS 2013, pp. 839–850. ACM, New York (2013). <http://doi.acm.org/10.1145/2508859.2516697>
8. Dolan-Gavitt, B., Leek, T., Zhivich, M., Giffin, J., Lee, W.: Virtuoso: narrowing the semantic gap in virtual machine introspection. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy, SP 2011, pp. 297–312. IEEE Computer Society, Washington, DC (2011). <https://doi.org/10.1109/SP.2011.11>
9. Dovgalyuk, P.: Deterministic replay of system’s execution with multi-target qemu simulator for dynamic analysis and reverse debugging. In: Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering, CSMR 2012, pp. 553–556. IEEE Computer Society, Washington, DC (2012)
10. Dovgalyuk, P., Dmitriev, D., Makarov, V.: Don’t panic: reverse debugging of kernel drivers. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 938–941. ACM, New York (2015). <http://doi.acm.org/10.1145/2786805.2803179>
11. Fu, Y., Lin, Z.: Space traveling across vm: automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In: 2012 IEEE Symposium on Security and Privacy (SP), pp. 586–600, May 2012
12. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings of the Network and Distributed Systems Security Symposium, pp. 191–206 (2003)
13. Guillon, C.: Program instrumentation with qemu. In: 1st International QEMU Users Forum, vol. 1, pp. 15–18 (2011)
14. Henderson, A., Prakash, A., Yan, L.K., Hu, X., Wang, X., Zhou, R., Yin, H.: Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 248–258. ACM, New York (2014). <http://doi.acm.org/10.1145/2610384.2610407>
15. Hizver, J., Chiueh, T.c.: Real-time deep virtual machine introspection and its applications. In: Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2014, pp. 3–14. ACM, New York (2014). <http://doi.acm.org/10.1145/2576195.2576196>
16. Julino, J.: Lightweight introspection for full system simulations. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, 1 March 2014. <http://os.itec.kit.edu/>
17. Lawton, K.P.: Bochs: A portable pc emulator for unix/x. *Linux J.* **1996**(29es) (1996). <http://dl.acm.org/citation.cfm?id=326350.326357>
18. Lefebvre, G., Cully, B., Head, C., Spear, M., Hutchinson, N., Feeley, M., Warfield, A.: Execution mining. *SIGPLAN Not.* **47**(7), 145–158 (2012). <http://doi.acm.org/10.1145/2365864.2151044>

19. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2005, pp. 190–200. ACM, New York (2005). <http://doi.acm.org/10.1145/1065010.1065034>
20. More, A., Tapaswi, S.: Virtual machine introspection: towards bridging the semantic gap. *J. Cloud Comput.* **3**(1), 1–14 (2014). <https://doi.org/10.1186/s13677-014-0016-2>
21. Tong, X., Moshovos, A.: Qtrace: a framework for customizable full system instrumentation. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 245–255, March 2015
22. Yan, L.K., Yin, H.: Droidscape: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium, Security 2012, p. 29. USENIX Association, Berkeley (2012). <http://dl.acm.org/citation.cfm?id=2362793.2362822>

# A Distributed Approach to Coreference Resolution in Multiagent Text Analysis for Ontology Population

Natalia Garanina<sup>(✉)</sup>, Elena Sidorova, and Irina Kononenko

A.P. Ershov Institute of Informatics Systems,  
Lavrent'ev av., 6, Novosibirsk 630090, Russia  
{garanina,lsidorova}@iis.nsk.su, irina\_k@cn.ru

**Abstract.** In the paper we describe a multi-agent ontology-based approach to coreference resolution and information extraction from input data. We define special class agents which correspond to ontology classes. They analyze current information in the corresponding retrieved instances. Results of this analysis are used for evaluation of instances' attributes, for detection of instances duplicates and equivalents, for fixing coreferential relations, and for assignment of worth of information connections used in disambiguation. The class agents act in the framework of our multi-agent approach to semantic text analysis for ontology population.

## 1 Introduction

The process of ontology population is considered as adding new instances of concepts to the ontology. This process is a part of ontology acquisition problem widely discussed, see e.g. [18]. A significant amount of the work concentrates on the task of knowledge acquisition from domain-specific content, which is mostly represented in natural language texts. In this context, the solution for the ontology population task is interrelated with the elaboration of natural language processing (NLP) techniques applied in the process of information extraction (IE). IE accomplishment implies several tasks, in particular, coreference resolution which is one of the most challenging NLP tasks.

In linguistics, reference is a relation of a text expression with some non-linguistic object or circumstances in the real or abstract world. Traditional linguistics considers two main classes of referential expressions: lexically full forms (names, nominal phrases, etc.) and reduced forms (e.g. pronouns). The coreference resolution problem is to identify a particular text mention of some non-linguistic entity with its other mentions in this text. The coreference resolution

---

The research has been supported by Russian Foundation for Basic Research (grant 15-07-04144, grant 17-07-01600) and Siberian Branch of Russian Academy of Science (Integration Grant n.15/10 "Mathematical and Methodological Aspects of Intellectual Information Systems").

has been a core research topic in computational linguistics since the 1960s, but the problem is far from being solved. Traditionally, the process of coreference resolution consists of two main stages: (1) detection of entity mentions that are probable candidates for coreference and (2) candidate mentions pairwise comparison to make the decision on candidate admissibility (whether the pair is valid or not). There are several basic approaches covered in the literature. The most important trends in the field have been found in the comprehensive surveys by R. Mitkov [16,17], and in [5,19]. These trends can be categorized into rule-based and learning approaches.

Early coreference resolution systems (dating back to 1970s and 1980s) are called “rule-based” (R. Mitkov) as they rely on hand-coded heuristics that specify whether two expressions can or cannot corefer [2,3,12,23]. The better term for this trend is “linguistic approach” [5] as it incorporates a lot of domain and linguistic knowledge: syntactic constraints, semantic features and preferences, discourse-oriented theories, such as Centering model [10], which can predict the focus of attention and choice of referring expression of a sentence. Theoretical models consider integrated knowledge sources and reveal factors that help to remove unlikely candidates until the minimal set of plausible candidates is obtained, and then make use of the center or focus, or other preferences. Modern theories investigating multiplicity of factors involved in the coreference phenomenon (such as the notion of Referent activation based on a discourse structure, antecedent syntactic or semantic role, animacy, etc. proposed in [13]) can be used directly or indirectly in work [14]. The mid-to-late 90’s gave rise to “corpus-based” (machine learning) approaches which were helped by the emergence of more powerful automatic parsers and taggers, and corpora annotated with coreference information to be used as a training data [11,22]. Paper [5] gives a survey of machine learning based techniques with respect to the coreference resolution task starting from a simple statistical naive Bayes-based model to methods using decision trees and conditional random fields and others. Unfortunately, in limited subject domains (for example, particular science and industrial domains) representative training text corpora do not exist usually. In this cases, it is reasonable to use classical rule-based methods.

There exist several attempts to apply distributed or agent-based techniques to the coreference resolution task: [20,24]. The known systems differ with respect to agent types and their functions. Thus, in [20] coreference resolution factors (recency, number agreement, gender agreement, animacy, disjoint reference, semantic consistency, global focus, cataphora and logical accessibility) are grouped in sets as constraint sources corresponding to the known partial theories of coreference. In [24] a common constraint agent allows for morphological agreement and semantic consistency and different coreference types (where an anaphor can be a name alias, appositional noun phrase, nominal predicate, definite noun phrase, demonstrative noun phrase, bare noun phrase) are charged to special agents. In both papers, agents corresponds to coreference resolution factors. Detection of coreference candidates is sequential. In [20] the agents process the decision about admissibility of the particular candidate in parallel, and in [24] the agents compose the system of sequential decision filters.

The proposed coreference resolution algorithm is used in our approach to text analysis and information extraction for population of a subject domain ontology. Our approach includes the following processes subsequently performed by the program modules: (1) the module of lexical analysis executes preliminary extraction of subject domain terms from a given text [15]; (2) the segmentator performs segmentation of the text into formal and genre fragments (sentences, sections, headlines, etc.) [21]; (3) the main analysis module constructs objects, corresponding to instances of a subject domain ontology, from the terms [7] and resolves coreference (this paper); (4) the disambiguation module resolves lexical and syntactic ambiguity [9]; (5) the population module updates the ontology with the processed objects (in plans). Our approach exploits the distributed multi-agent architecture of text processing [1, 4, 6] as opposed to the generally accepted sequential NLP procedure. This multi-agent approach to NLP is both motivated from the semantic point of view and computationally effective and robust: the agents speed up processing since they act in parallel, and they efficiently use data resource exactly when and where this is needed. Our agents are associated with vocabulary terms, ontology classes, ontology instances, and rules of ontology population. Sequential data analysis is less efficient, since it suggests sorting rules to information extraction. The agent-based approach allows one to get rid of sorting rules.

In our framework the coreference resolution problem is to detect if some group of retrieved objects refer to the particular ontology instances. Our approach to the coreference resolution is rule-based because we deal with limited subject domains. The proposed algorithm is ontology-driven as it strongly relies on the structure of the underlying predefined domain ontology. We focus on full lexical items (nominals and names), as they bear more semantic clues than pronominals for making comparisons with ontology classes and instances. Ambiguities occurring at the linguistic level are resolved at the ontology level. We use a similarity measure to compare potential coreferent objects within the group. The evaluation of the measure integrates textual factors (such as a text distance and context dependence) with factors based on ontological properties of instances' attributes (class hierarchy, composition, transitivity, etc.). This way of using the ontology structure allows one to resolve coreference more precisely even for complex objects such as descriptions of events and situations presented as ontology polyadic relations. To the best of our knowledge, coreferencing such complex objects is non-trivial and investigated deficiently. Another advantage of our approach is that coreference relations between the candidate objects are fixed during information extraction. The corresponding candidate agents update their information using these relations. This update let us recognize ontology instances and relations faster and more completely than in the traditional approaches. The class agents detect and resolve the coreference candidates using the similarity measure. They act in parallel mode which speeds up the process in comparison with the sequential and multi-agent approaches mentioned above.

The rest of the paper is organized as follows. In Sect. 2, we give background definitions and formally state the problem of coreference resolution. Section 3



outlines the process of multi-agent text analysis, and gives the detailed description of the class agents and their protocols for coreference resolution. In the concluding Sect. 4, directions of future research are discussed.

## 2 Problem Statement and Base Definitions

Let us consider an ontology of a subject domain, the ontology population rules, the semantic and syntactic models for the language of the subject domain, the term vocabulary, and input data as a finite natural language text with information for population of the ontology. We assume that *an ontology  $O$  of a subject domain* includes the following elements: (1) the finite nonempty set  $C_O$  of *classes* for concepts of the subject domain, (2) the finite set  $D_O$  of *data domains*, and (3) the finite set of *attributes* with names in  $Dat_O \cup Rel_O$ , each of which has values in some data domain from  $D_O$  (*data attributes* in  $Dat_O$ ) or has values as instances of some classes (*relation attributes* in  $Rel_O$ , which model relations). Every class  $c \in C_O$  is defined by the tuple of attributes:  $c = (Dat_c, Rel_c)$ , where every data attribute  $\alpha \in Dat_c \subseteq Dat_O$  has the domain  $d_\alpha^c \in D_O$  with values in  $V_{d_\alpha^c}$  and every relation attribute  $\rho \in Rel_c \subseteq Rel_O$  has values from classes  $c_\rho \subseteq C_O$ . The set of all class attributes is  $Atr_c = Dat_c \cup Rel_c$ . For an attribute  $\gamma$  of a class  $c$ , let  $c^\gamma = c$  and  $D^\gamma$  be the set of its values. The set of class attributes includes the nonempty set of *key attributes*  $Atr_c^K$ . The key attributes can be the data as well as the relation attributes. We say that  $a$  is *an instance of the class*  $c_a = (Dat_{c_a}, Rel_{c_a})$  ( $a \in c_a$ ) iff  $a = (c_a, Dat_a, Rel_a)$ , where every data attribute in  $Dat_a$  has a name  $\alpha_a \in Dat_{c_a}$  with the values  $V_{\alpha_a}$  from  $V_{d_{\alpha_a}^{c_a}}$  and every relation attribute in  $Rel_a$  has a name  $\rho_a \in Rel_{c_a}$  with the values  $V_{\rho_a}$  as instances of the classes from  $c_{\rho_a}$ . The data key attributes are always one-valued, i.e. every of key attribute of every ontology instance can have only single value. The relation key attributes correspond to bijective relations. We consider an ontology without data and class synonyms, i.e.  $\forall \alpha_1, \alpha_2 \in Dat_O : d_{\alpha_1} \neq d_{\alpha_2}$  and  $\forall c_1, c_2 \in C_O : Atr_{c_1} \neq Atr_{c_2}$ . *An information content  $IC_O$  of the ontology  $O$*  is a set of instances of the classes from  $O$ . *The ontology population problem* is to compute an information content for a given ontology from given input data.

Let us define a set  $A$  of *informational objects* (*i-objects*) retrieved from input data and corresponding to ontology instances. Every informational object  $a \in A$  has the form  $(c_a, Dat_a, Rel_a, G_a, P_a)$ , where

- the class  $c_a \in C_O$ ;
- $Dat_a$  is the set of data attribute  $\alpha_a = (\alpha, Val_{\alpha_a})$ , where
  - the name  $\alpha \in Dat_{c_a}$ , and
  - $Val_{\alpha_a}$  is the set of information values  $\bar{v} = (v_{\bar{v}}, s_{\bar{v}})$  with
    - the data value  $v_{\bar{v}} \in d_{\alpha_a}^{c_a}$ , the set of values of  $\alpha_a$  is  $V_{\alpha_a} = \{v_{\bar{v}} | \bar{v} \in Val_{\alpha_a}\}$ ,
    - $s_{\bar{v}}$  is structural information (a position in input data),
- $Rel_a$  is the set of relation attribute  $\rho_a = (\rho, V_{\rho_a})$ , where
  - the name  $\rho \in Rel_{c_a}$ , and
  - $V_{\rho_a}$  is the set of i-objects of a class  $c_{\bar{\rho}}$  from  $c_{\rho_a}$ ;

- $G_a$  is grammar information (morphological and syntactic features);
- $P_a$  is structural information (a set of positions in input data).

$Atr_a = Dat_a \cup Rel_a$  is the set of all attributes. Note, that features of natural language processing could cause assigning one-valued (key, in particular) attributes of i-objects with many values. These ambiguities is resolved after the coreference resolution. Every i-object corresponds to some ontology instance in a natural way. Let  $a = (c_a, Dat_a, Rel_a, G_a, P_a)$  be an i-object, then its corresponding ontology instance is  $a' = (c_a, Dat_{a'}, Rel_{a'})$ , and every  $\alpha \in Dat_{a'}$  has value(s) in  $V_{\alpha_a}$  and every  $\rho \in Rel_{a'}$  has values in  $V_{\rho_a}$ .

For the description of the process of the coreference resolution, we introduce the following collative relations on i-objects  $a, b \in A$ .

- *Duplication*:  $a$  and  $b$  are duplicates ( $a = b$ ) iff  $Atr_a^K = Atr_b^K$ , and  $P_a = P_b$ .
- *Ontological equivalence*:  $a$  and  $b$  are ontological equivalents ( $a \equiv b$ ) iff  $Atr_a^K = Atr_b^K$ , and  $P_a \neq P_b$ .
- *Coreference*:  $a$  and  $b$  are coreferents ( $a \approx b$ ) iff  $Atr_a^K \subseteq Atr_b^K \vee Atr_b^K \subseteq Atr_a^K$ , where  $Atr_a^K \subseteq Atr_b^K$  iff  $\forall \gamma_a \in Atr_a^K : V_{\gamma_a} \neq \emptyset \rightarrow (\exists \delta_b \in Atr_b^K : \gamma_a \subseteq \delta_b)$ , where  $\gamma_a \subseteq \delta_b$  iff  $(\gamma_a, \delta_b \in Dat_O \wedge V_{\gamma_a} \subseteq V_{\delta_b}) \vee (\gamma_a, \delta_b \in Rel_O \wedge V_{\gamma_a} \subseteq^r V_{\delta_b})$ , where  $\subseteq^r$  is defined in the next paragraph.

The following definitions concern the coreference. Let  $a, b, c \in A$ ,  $X, Y \subseteq A$ .

- *The coreferential group of the i-object  $a$  (co-group)* is  $coR(a) = \{x \in A \mid x \approx a\}$ .
- *The co-group of the set  $X$*  is  $coR(X) = \bigcup_{x \in X} coR(x)$ .
- *Coreferential membership*:  $a \in^r X$  iff  $a \in coR(X)$ .
- *Coreferential inclusion*:  $X \subseteq^r Y$  iff  $\forall x \in X : x \in^r Y$ .
- *Coreferential intersection*:  $X \cap^r Y = coR(X) \cap coR(Y)$ .
- *Coreferential conflict*: i-objects  $a$  and  $b$  are in the coreferential conflict with respect to i-object  $c$  ( $a \overset{c}{\curvearrowright} b$ ) iff  $a \approx c \wedge b \approx c \wedge a \notin coR(b)$ . The coreferential conflict means that some i-object is coreferent to two non-coreferential i-objects.

*The coreference resolution problem* is to detect if given i-objects (*coreferents or candidates for coreference*) correspond to the same ontology instance. Our algorithm for coreference resolution constructs conflict-free co-groups. This construction uses *coreference similarity* of i-objects for resolving coreferential conflicts. The measure of the coreference similarity consists of semantic, context, position and grammar measures.

We formulate some properties of the ontology elements which are used in the processes of detection and resolution of coreferences. All properties except a refinement relation are known in the ontology and description logics framework.

- *The class inheritance relation*. A class  $c_2$  inherits a class  $c_1$  ( $c_1 < c_2$ ) iff  $\forall a \in c_2 : a \in c_1$ .
- *The sub-attribute relation*. An attribute  $\gamma_2$  is a sub-attribute of an attribute  $\gamma_1$  ( $\gamma_2 \ll \gamma_1$ ) iff  $\forall a \in c^{\gamma_2} : v \in V_{\gamma_2 a} \rightarrow v \in D^{\gamma_1}$ .

For relation attributes:

- The ternary *composition relation*. An attribute  $\rho$  is a composition of attributes  $\rho_1$  and  $\rho_2$  ( $\rho = \rho_1 \circ \rho_2$ ) iff  $\forall a \in c^\rho : a' \in V_{\rho_1 a} \wedge a'' \in V_{\rho_2 a'} \rightarrow a'' \in V_{\rho a}$ .
- The *refinement relation*. An attribute  $\rho$  refines an attribute  $\rho'$  ( $\rho \triangleright \rho'$ ) iff  $\forall a \in c^\rho, a' \in c^{\rho'} : V_{\rho a} \cap V_{\rho' a'} \neq \emptyset \rightarrow a' \in V_{\rho a}$ .
- The *transitivity*. An attribute  $\rho$  is transitive ( $\rho \in Rel_O^t$ ) iff  $\forall a_1, a_2, a_3 \in c^\rho : a_2 \in V_{\rho a_1} \wedge a_3 \in V_{\rho a_2} \rightarrow a_3 \in V_{\rho a_1}$ .

First, we define the *semantic measure* of the coreference similarity. This measure takes into account the attribute similarity of i-objects. For the data attributes, the power of this similarity is defined by the number of equal values of the attributes (item 1). For the relation attributes, we consider four types of similarity. The power of the standard relation similarity is also defined by the number of equal values of the attributes according to coreferences of the values (item 2). The power of the composition similarity is defined by the number of equal values of the attributes which are in a composition relation with some of the values according to coreferences of the values (item 3). The power of the refinement similarity is defined by the number of pairs of values of the attributes which are in a refinement relation with some of the values according to coreferences of the values (item 4). The power of the transitive similarity is defined by the number of pairs of values of attributes which are in a transitive relation according to coreferences of the values (item 5). The semantic similarity is defined by the normalized sum of all above powers of similarity. The formal definitions follow.

Let  $a, b \in A$ ,  $a \approx b$ , and  $\alpha_a \in Dat_a, \beta_b \in Dat_b, \rho_a \in Rel_a, \xi_b \in Rel_b$ .

- (1)  $\alpha_a$  is *data similar* to  $\beta_b$  ( $\alpha_a \sim_d \beta_b$ ) iff  $S = V_{\alpha_a} \cap V_{\beta_b} \neq \emptyset$ ;  
The power of d-similarity is  $sim(\alpha_a, \beta_b) = \frac{|S|}{2} \left( \frac{1}{|V_{\alpha_a}|} + \frac{1}{|V_{\beta_b}|} \right)$ .
- (2)  $\rho_a$  is *relation similar* to  $\xi_b$  ( $\rho_a \sim_{rl} \xi_b$ ) iff  $S = V_{\rho_a} \cap^r V_{\xi_b} \neq \emptyset$ ;  
The power of rl-similarity is  $sim(\rho_a, \xi_b) = \frac{|S|}{2} \left( \frac{1}{|coR(V_{\rho_a})|} + \frac{1}{|coR(V_{\xi_b})|} \right)$ .
- (3)  $\rho_a$  is *composition similar* to  $\xi_b$  ( $\rho_a \sim_c \xi_b$ ) iff  $\exists \pi \in Rel_O : \xi = \rho \circ \pi$  and  $S^c = \{o | o \in c^\pi \text{ and } o \in^r V_{\rho_a} \wedge V_{\xi_b} \cap^r V_{\pi_o} \neq \emptyset\} \neq \emptyset$ .  
The power of c-similarity is  $sim(\rho_a, \xi_b) = \frac{|S^c|}{|coR(V_{\rho_a})|}$ .
- (4)  $\rho_a$  is *refinement similar* to  $\xi_b$  ( $\rho_a \sim_r \xi_b$ ) iff  $\exists \pi \in Rel_O : \rho \triangleright \pi$  and  $S^r = \{(o, p) | o \in^r V_{\rho_a}, p \in V_{\xi_b} \cap^r V_{\pi_o}\} \neq \emptyset$ .  
The power of r-similarity is  $sim(\rho_a, \xi_b) = \frac{|S^r|}{|coR(V_{\rho_a})| \cdot |coR(V_{\xi_b})|}$ ;
- (5)  $\rho_a$  is *transitive similar* to  $\xi_b$  ( $\rho_a \sim_t \xi_b$ ) iff  $\rho = \xi \wedge \rho \in Rel_O^t$  and  $S^t = \{(o, p) | o \in^r V_{\rho_a}, p \in^r V_{\xi_b} \text{ and } p \in^r V_{\rho_o} \vee o \in^r V_{\rho_p}\} \neq \emptyset$ .  
The power of t-similarity is  $sim(\rho_a, \xi_b) = \frac{|S^t|}{|coR(V_{\rho_a})| \cdot |coR(V_{\xi_b})|}$ ;  
The power of semantic similarity:  $S(a, b) = \frac{1}{|Sim_a^b|} \sum_{(\gamma_a, \delta_b) \in Sim_a^b} sim(\gamma_a, \delta_b)$ ,  
where  $Sim_a^b = \{(\gamma_a, \delta_b) | \gamma_a \in Atr_a, \delta_b \in Atr_b \text{ and } sim(\gamma_a, \delta_b) \neq 0\}$  is the set of similar attributes.

The following proposition states the necessary conditions for the semantic similarities of i-objects. The proof immediately follows from the definitions of

the ontology elements properties. In the proposition, *hierarchical intersection*  $\cap^i$  for sets of classes  $C_1$  and  $C_2$  is  $C_1 \cap^i C_2 = \{c \mid c \in C_j \wedge \exists c' \in C_k : c \leq c' \vee c \geq c', j, k \in \{1, 2\} \wedge j \neq k\}$ , and *hierarchical inclusion*  $\subseteq^i$  is defined as  $C_1 \subseteq^i C_2$  iff  $\forall c \in C_1 \exists c' \in C_2 : c \leq c' \vee c \geq c'$ .

**Proposition 1.** *Let  $a, b \in A$ ,  $\alpha_a \in Dat_a$ ,  $\beta_b \in Dat_b$ ,  $\rho_a \in Rel_a$ , and  $\xi_b \in Rel_b$ .*

- (1) *Coreference:*  $a \approx b \Rightarrow c_a = c_b \vee c_a < c_b \vee c_a > c_b$ .
- (2) *d-similarity:*  $\alpha_a \sim_d \beta_b \Rightarrow \alpha = \beta \vee \alpha \ll \beta \vee \alpha \gg \beta$ .
- (3) *rl-similarity:*  $\rho_a \sim_{rl} \xi_b \Rightarrow \rho = \xi \vee \rho \ll \xi \vee \rho \gg \xi \vee c_\rho \cap^i c_\xi \neq \emptyset$ .
- (4) *c-similarity:*  $\rho_a \sim_c \xi_b$  and  $\xi = \rho \circ \pi \Rightarrow c^\xi \cap^i c^\rho \neq \emptyset \wedge \{c^\pi\} \subseteq^i c^\rho \wedge c_\xi \subseteq^i c_\pi$ .
- (5) *r-similarity:*  $\rho_a \sim_r \xi_b$  with  $\rho \triangleright \pi \Rightarrow \{c^\xi\} \subseteq^i c_\rho \wedge \{c^\pi\} \subseteq^i c_\xi \wedge c_\pi \subseteq^i c_\rho$ .
- (6) *t-similarity:*  $\rho_a \sim_t \xi_b$  with  $\rho \in Rel_O^t \Rightarrow \{c^\rho\} \subseteq^i c_\rho$ .

Second, we define *the context measure*. This measure takes into account information connectivity of i-objects in a given text. The process of constructing i-objects generates directed information connections between the i-objects when the attribute values of one i-object is used for evaluation of the attributes of another i-object. If an i-object  $x$  uses information from an i-object  $y$  (may be through other i-objects) then  $x$  is *an information descendant of  $y$* :  $x \in Des(y)$ . Let after finish of the construction process coreferent i-objects  $a$  and  $b$  have common descendants and  $CD(a, b) = \sum_{x \in Des(a) \cap Des(b)} cw(x) + 1$ , where  $cw(x)$  is the context weight of the i-object  $x$ . One could use the adopted Finding Relatives algorithm from [9] to compute  $CD(a, b)$ . *The power of common descendants* is  $D(a, b) = 1 - \frac{1}{CD(a, b)}$ . The other component of the context measure is *a context weight of attributes* from a given coreferent. In the process of the construction an i-object can borrow values of attributes from its coreferents. The power of this borrowing depends on the number of descendants which use these attribute values. The context weight  $cw(\gamma)$  for the borrowed attribute  $\gamma \in Atr_a$  could be computed by the adopted Weight Computing algorithm from [9]. The overall context weight of borrowed attributes is  $CW(a, b) = \sum_{\gamma \in Br(a, b)} cw(\gamma) + 1$ , where  $Br(a, b)$  are attributes which values are borrowed by  $a$  from  $b$ . Note that this measure is asymmetric:  $CW(a, b) \neq CW(b, a)$  because  $Br(a, b) \neq Br(b, a)$ . *The power of borrowed attributes* is  $B(a, b) = 1 - \frac{1}{CW(a, b) + CW(b, a)}$ . Now, *the power of context similarity* of i-objects  $a$  and  $b$  is  $C(a, b) = \frac{1}{2}(D(a, b) + B(a, b))$ .

Third, we define *the position measure*. This measure takes into account various forms of closeness of i-objects in an input text. Components of the position measure are *the segment measure*  $TS(a, b)$ , *the ambiguous coreferent measure*  $CO(a, b)$ , and *the lexeme measure*  $L(a, b)$ . *Segments* are named text position ranges predefined by the segmentator. A segment can be a sentence, a paragraph, a title, a section, a list, etc. Let a segment  $F$  be the text scope of coreferent searching. This segment is composed of  $n$  segments  $s_i$  which are used for measuring the distance between i-objects' positions:  $F = s_1 s_2 \dots s_n$ . For example,  $F$  could be a paragraph and  $s_i$  could be embedded sentences. Let the position  $P_a$  of the i-object  $a$  be in  $s_i$  and the position  $P_b$  of the i-object  $b$  be in  $s_j$ . Then  $TS(a, b) = 1 - \frac{|i-j|}{n}$ . The ambiguous coreferent measure takes into

account the number of i-objects which are in the coreferential conflict with  $b$  with respect to  $a$ . Let  $P_b < P_a$  and  $acb(a, b) = \{c \mid c \overset{a}{\rightsquigarrow} b \wedge P_b < P_c < P_a\}$ . Then  $AC(a, b) = \frac{1}{|acb(a, b)|+1}$ . The lexeme measure is simply the number of text lexemes between the positions of  $a$  and  $b$ . Let  $lob(a, b) = \{l \mid P_b < P_l < P_a\}$ . Then  $L(a, b) = \frac{1}{|lob(a, b)|+1}$ . Now the power of position similarity of i-objects  $a$  and  $b$  is  $P(a, b) = \frac{1}{3}(TS(a, b) + AC(a, b) + L(a, b))$ .

Fourth, we define the grammar measure based on the standard linguistic features such as gender, number, person, tense, etc. We do not go into details in this paper. Let the power of grammar similarity be  $G(a, b) \leq 1$ .

Now we define the measure of similarity of coreferential i-objects  $a$  and  $b$  as  $w(a, b) = \frac{1}{4}(S(a, b) + C(a, b) + P(a, b) + G(a, b))$ . In future work, we plan to estimate the contribution of each component to this measure by experiments and to use the corresponding coefficients in this formula. If  $a \overset{c}{\rightsquigarrow} b$  we consider that the coreferential conflict is resolved to  $a$  iff  $w(a, c) > w(b, c)$ .

### 3 The Coreference Resolution in the Multiagent Information Extraction

The outline of our approach to text semantic analysis and information extraction for ontology population follows. The preliminary stage of text processing is executed by the module of lexical analysis based on the vocabulary of the subject domain. This module constructs the terminological cover consisting of lexical objects which are tagged terms found in the text. The segmentator-module produces the segment cover which represents text decomposition into formal and genre subunits. The terminological cover is a basis for creating and updating i-objects. The lexical objects are used for evaluating attributes of i-objects. The analysis rules implement language processing rules and ontology population rules. They are formulated by experts taking into account the ontology and sublanguage of subject domain. The main analysis module implements the following processes of constructing i-objects. The analysis rules generate new information based on information (attribute values) taken from i-objects and lexical objects. The rules use this information to define new attribute values of existing i-objects and to generate new i-objects. Every rule can take information only from its corresponding sets of i-objects which must be linguistically and ontologically compatible. Using information by some rule from one i-object for other i-object sets the information connection between these i-objects labeled by this information. These connections keep the history of an i-object. They are used for evaluation of integration of an i-object into a text. These constructing processes terminate when the analysis rules cannot generate new information.

In the process the following problems arise: (1) the coreference, i.e. associating different i-objects to the same ontology object; (2) lexical ambiguity, when different i-objects correspond to the same text fragment; (3) syntactic ambiguity, which means incorrect evaluation of attributes as a result of variety of syntactic interpretations. Our method of lexical and syntactic disambiguation is based on computing and comparing the context cardinalities and evidence powers of

i-objects [9]. For this method to be correct, the coreference resolution must be performed before the disambiguation. The coreference resolution is the construction of the conflict-free co-groups of i-objects. The algorithm of constructing the co-groups is performed by the main analysis module in parallel with constructing i-objects. The coreferential conflict resolution in the co-groups is based on the similarity of i-objects. This resolution is performed by the main analysis module after termination of constructing i-objects. If it has found that i-objects under consideration are coreferent then their attribute values and information connections have to be joined.

We associate every i-object, every analysis rule, and every ontology class with different agents which perform different tasks of text analysis for ontology population: (1) creating and updating i-objects, (2) coreference resolution, and (3) ambiguity resolution. The corresponding i-agents, rule agents, and class agents communicate and interchange data for executing the tasks. There is also an auxiliary agent: the agent-master detects termination and coordinates agents in the disambiguation process. For the details of the first and the last tasks, see [7,9]. The result of agent interactions is the maximally determined system of i-objects without the coreference, lexical, and syntactical ambiguities. All agents execute their protocols in parallel until it happens that none of the agents can proceed. These termination events are determined by the master agent. We use our original algorithm for termination detection which is based on activity counting [8]. The system is dynamic because the rule agents can create new information agents, the class agents can kill the i-agents by joining duplicates and ontological equivalents, and the master agent can kill the i-agents whose i-objects are weakly integrated in a given text. The agents are connected by duplex channels. The master agent is connected with all agents, the i-agents are connected with their rule agents, class agents, and successors/predecessors by information connections. Messages are transmitted instantly via a reliable medium and stored in channels until being read. In this paper we describe protocols for the class agents and some adaptations for the protocols of the i-agents and rule-agents from [7] which we use to resolve the coreference problem. We give semiformal definitions of the class agent and its protocols for performing this task.

Every *i-agent* from the set  $IA$  corresponds to some i-object from  $A$ . For an i-agent  $I$ , the class  $c_I$ , set  $Atr_I$ , and position  $P_I$  are the class, attributes, and position of its i-object  $a_I$ . We omit other data of i-agents unnecessary for the topic of the paper. We describe the class agents in detail. Each *class agent*  $C$  is the tuple  $C = (c_C, P_C, Sg_C, S_C, R_C)$ , where

- $c_C$  is the class of the agent;
- $Rlt_C = \{P_C\} \cup Ch_C$  are the relatives of the agent, where
  - $P_C$  is the predecessor class agent  $C'$  ( $c_{C'} > c_C$ );
  - $Ch_C = \{C' \mid c_{C'} < c_C\}$  is the set of successor class agents;
- $Sg_C$  is the segment condition for coreference scopes and distances;
- $S_C$  is the set of scope segments  $s \in S_C$  with i-agents in:  $s = (P_s, A_s)$ , where
  - $P_s$  is the range of the text position of the segment and
  - $A_s$  is the set of i-agents of the class  $c_C$  inside  $P_s$ ;

- $R_C$  is the set of co-groups of i-agents, where every  $r_I \in R_C$  has the form  $r_I = (I, CR_I, NCR_I, W_I)$ , where
  - $I$  is the i-agent,
  - $CR_I \subseteq IA$  is the set of coreferents,
  - $NCR_I \subseteq IA$  is the set of rejected coreferents, and
  - $W_I$  is the set of similarity measures  $w(I, J)$  for  $J \in CR_I$ .

First adaptation for the i-agent protocol is that every i-agent  $I$  has to send its id to the corresponding class agent with  $c_C = c_I$  immediately after its activation. Second adaptation is that every i-agent sends the special signal to its class agent immediately after update its attribute values. The main novelty of the adaptation is that i-agents can use the attribute values of their coreferents as their own attribute values in their communication with the rule agents. The i-agents and rule agents attach the history of attribute's origin to every sending attribute values. The brief descriptions of protocols for the class agents constructing the conflict-free co-groups follow.

**(1) The interface protocol for the class agents.** This protocol specifies agent's reactions for incoming messages. These messages include information which actions should be performed by the agent:

- (1) (**Start**): to start, sent by the master agent;
- (2) (**Create, I**): to create the co-group for the new i-agent  $I$  by the protocol **Create(I)**, sent by the i-agent  $I$ ;
- (3) (**Update, I**): to update the co-group for the existing i-agent  $I$  by the protocol **Update(I)**, sent by the i-agent  $I$ ;
- (4) (**Replace, I, J**): to replace the i-agent  $J$  by the i-agent  $I$  in the class co-groups by the protocol **Replace(I, J)**, sent by the parent or child class agent (the protocol is trivial and not described in the paper);
- (5) (**ResConf**): to compute the similarity measures by the protocol **Sim()** and to resolve coreferential conflicts by the protocol **Resolve()**, sent by the master;
- (6) (**UpdFin, R**): to update its co-groups with conflict-free co-group  $R$  by the protocol **UpdFin(R)**, sent by the child class agent.

Until an input message causes the agent to react the agent stays in a wait mode. Messages for the agent are stored in its input channel.

In the protocols, the procedure **join(I, J)** (1) returns one i-agent made of two i-agents by making one i-object of the lower class from two corresponding i-objects via joining their attribute values, grammar and position information, and combining other data of the i-agents; (2) joins co-groups of the agents in the co-group of the returning agent; (3) tries to expand the co-group of the new agent by the procedure **ExpandRef** described below. We use the notation  $Sg_C^I$  for the coreference scope of the agent  $I$  with respect to the class  $C$ . In the following,  $X.add(Y)$  and  $X.rem(Y)$  denote adding/removing the elements and sets  $Y$  to/from the set  $X$ , respectively. If attribute values are received by i-agent from some rule agent we call them *the own attribute values*. In other case, if attribute values are borrowed by i-agent from some another i-agent we call them *the borrowed attribute values*.

## (2) Creating co-groups

First, the class agent  $C$  checks if the coreference scope of  $I$  is equal to some existing coreference scope. Depending on the result it adds  $I$  to the existing scope (line 1) or creates a new scope and fills it with the i-agents which positions are in this scope (line 2). The new co-group for the agent  $I$  is created in line 3. The class agent does not perform the pairwise collative testing for the pairs of low class i-agents (line 5), because it is performed by some low class agent and the result is stored by the corresponding agent. The agent  $C$  tests the i-agent  $I$  and agents from the scope segment for duplication, equivalence, and coreference. In this testing the agent use only the own attribute values of i-agent. If they are duplicates or equivalents then they are joined, and  $C$  returns to wait mode (line 6). If  $I$  meets a coreferent then the class agent adds the coreferent to the co-group of  $I$  (line 7). At line 8 the class agent tries to find equivalents for  $I$  among all i-agents having co-groups stored by this class agent, because the search of equivalents is not restricted by a special segment. Then the co-group of  $I$  is expanded by coreferents which are not from its scope, but in the scope of its coreferents (line 9). In line 10 every co-group of coreferents is also expanded by coreferents which are not from their scope, but in the scope of their coreferent  $I$ . After that, the co-group of  $I$  is added to the set of co-groups  $R_C$ . Finally, the class agent  $C$  sends the i-agent  $I$  to its parent class agent because there could be coreferents for  $I$  at the next level of the ontology hierarchy (line 12).

Create( $I$ ) ::

1. if  $\exists s' \in S_C : Sg_C^I = P_{s'}$  then  $A_{s'}.add(I)$ ;  $s = s'$ ;
2. else  $s = (Sg_C^I, \{I\})$ ;  
     forall  $r_J \in R_C$  if  $P_J \in P_s$  then  $A_s.add(J)$ ;  
      $s_C.add(s)$ ;
3.  $r_I = (I, \{I\}, \emptyset, \emptyset)$ ;
4. forall  $J \in A_s$
5.     if  $c_I \neq c_C \wedge c_J \neq c_C$  then continue;
6.     if  $I = J \vee I \equiv J$  then  $J = join(J, I)$ ; kill( $I$ ); return;
7.     if  $I \approx J$  then  $CR_I.add(J)$ ;
8. forall  $r_J \in R_C \setminus CR_I$   
     if  $c_I \neq c_C \wedge c_J \neq c_C$  then continue;  
     if  $I \equiv J$  then  $J = join(J, I)$ ; kill( $I$ ); return;
9. forall  $J \in CR_I$  ExpandRef( $I, J$ );
10. forall  $J \in CR_I$  ExpandRef( $J, I$ );
11.  $R_C.add(r_I)$ ;
12. send  $I$  to  $P_C$ ;

The following procedure can expand the coreference relation of the i-object of the i-agent  $I$  to i-objects from other scope segments of the agents from co-group of the i-agent  $J$ . The coreferent i-agents include each other into their co-groups (line 4), and exchange by the own attribute values labeled by 0 (lines 5–7). If an i-agent receives some attribute value from its coreference  $I$  then this attribute value is labeled by  $I$ . For describing this, we use the notation:  $Atr(X/Y)$  is the set of attribute values in which the label  $X$  is changed for  $Y$ .



ExpandRef( $I, J$ ) ::

1. forall  $M \in CR_J$
2.   if  $M \in CR_I \vee (c_I \neq c_C \wedge c_M \neq c_C)$  then continue;
3.   if  $I \approx M$  then
4.      $CR_I.add(M); CR_M.add(I);$
5.     forall  $J \in CR_I$  send( $Atr_M(0/IM)$ ) to  $J$ ;
6.     send( $Atr_M(0/JM)$ ) to  $I$ ;
7.     send( $Atr_I(0/IJ)$ ) to  $M$ ;

### (3) Updating co-groups

If the key attribute values of the i-agent  $I$  is updated by some rule agent then it is necessary to test the agent  $I$  and its coreferents for the collative relations again. This checking for the low class i-agents is performed by these low class agents (line 4). If the i-agent  $I$  becomes to duplicate or equivalent to some coreferent, the class agent joins them and other relative class agents replaces the coreferent by  $I$  (line 5). Update of key attribute values can causes the co-group reduction (line 6). Again, equivalence to other i-agents outside the coreference scope of  $I$  is checked (lines 7–9). The new values should be sent to all coreferents (line 10).

Update( $I$ ) ::

1. send  $I$  to  $P_C$ ;
2. if upd\_key( $Atr_I$ ) then
3.   forall  $J \in CR_I$
4.     if  $c_I \neq c_C \wedge c_J \neq c_C$  then continue;
5.     if  $I = J \vee I \equiv J$  then
6.        $I = \text{join}(I, J)$ ; send (Replace,  $I, J$ ) to  $Rlt_C$ ; kill( $J$ );
7.     if  $\neg(I \approx J)$  then  $CR_I.rem(J); CR_J.rem(I)$ ;
8.     forall  $r_J \in R_C \setminus CR_I$
9.       if  $c_I \neq c_C \wedge c_J \neq c_C$  then continue;
10.       if  $I \equiv J$  then
11.          $I = \text{join}(I, J)$ ; send (Replace,  $I, J$ ) to  $Rlt_C$ ; kill( $J$ );
12. forall  $J \in CR_I$  send( $Atr_I(0/I)$ ) to  $J$ ;

### (4) Computing the similarity measure

The class agent  $C$  computes the similarity measure for a pair of i-agents if one of them is in the class  $c_C$ . The computation for low class i-agents is performed by these low class agents. The function computeW uses the formula of the similarity measure from the previous section. The computed measures are stored in the corresponding co-groups.

Sim() ::

w real;

1. forall  $r \in R_C$
2.    $I = I_r$ ;
3.   forall  $J \in CR_I$
4.     if  $c_I \neq c_C \wedge c_J \neq c_C$  then continue;
5.     if  $w(I, J) \in W_I$  then continue;
6.      $w = \text{computeW}(I, J)$ ;
7.      $W_I.add((w, J)); W_J.add((w, I));$

### (5) Resolving coreferential conflicts

The coreferential conflict resolution uses the computed similarity measures and the resolution results from the low class agents. Hence the class agent  $C$  waits finish of coreference resolution by all its child classes (line 1). Further the agent resolves the conflicts by comparing similarities (line 5). The class agent removes the more distant coreference from the corresponding co-groups and records non-coreferents (lines 6, 7). If non-coreferential i-agents have been exchanged with attribute values then these values should be removed from the corresponding i-objects and their descendants (lines 8, 9). For every agent from the obtained conflict-free co-group  $CR_I$  of the agent  $I$ , its conflict-free co-group is exactly equal to  $CR_I$  due to the expansion by the procedure `ExpandRef` and our way of the conflict resolution. Hence it is reasonable to remove these co-groups without processing (line 10). At the end, the class agent sends to its parent class agent the result of coreference resolution as its set of conflict-free co-groups.

```
Resolve() ::
1. wait (finish_Resolve);
2. forall  $r \in R_C$ 
3.    $I = I_r$ ;
4.   forall  $J_1, J_2 \in CR_I : J_1 \notin CR_{J_2}$ 
5.     if  $w(I, J_1) < w(I, J_2)$  then  $J = J_1$ ; else  $J = J_2$ ;
6.      $CR_I.\text{rem}(J)$ ;  $NCR_I.\text{add}(J)$ ;
7.      $CR_J.\text{rem}(I)$ ;  $NCR_J.\text{add}(I)$ ;
8.     forall  $\gamma^J \in Atr_I$  remove( $\gamma^J$ );
9.     forall  $\gamma^I \in Atr_J$  remove( $\gamma^I$ );
10.  forall  $J \in CR_I$   $R_C.\text{rem}(r_J)$ ;
11. send (UpdFin,  $R_C$ ) to  $P_C$ ;
```

When the class agent  $C$  receives the coreference results from its child agent  $C'$ , it removes all non-coreferents from the corresponding co-groups.

```
UpdateFin( $R_{C'}$ ) ::
1. forall  $r_{I'} \in R_{C'}$ 
2.   for  $r_I \in R_C$  with  $I = I'$ 
3.      $CR_I.\text{rem}(NCR_{I'})$ ;  $NCR_I.\text{add}(NCR_{I'})$ ;
```

When the class agents finish the coreferential conflict resolution, the conflict-free groups of coreferent i-objects are constructed. Further the process of lexical and syntactical disambiguation takes into account the attribute values from the coreferents and the similarity measures of the coreferents for estimation of integration of the i-objects into a given text. We use the similarity measures as the weight of the information connections in formulas of the context weight of i-objects (see [9]).

## 4 Conclusion

In this paper we suggest the approach to coreference resolution in the framework of multi-agent text analysis for ontology population. This approach takes into account semantic, textual and grammatical similarity of coreferent objects.

These similarities are integrated into the single estimation of coreferential similarity. One of the novelty of the approach is that semantic similarity is based on properties of ontology classes and relations such as hierarchy, composition, refinement, and transitivity which give more precise and complete coreference identification. The multi-agent aspect of the approach is that the special agents-classes create coreferential groups of objects corresponding to the ontology instances because the necessary criterion of coreference is (hierarchical) equality of ontology classes of the objects. Besides, our approach allows one to resolve reference of (n-ary) ontology relations represented as special ontology classes. The agents-classes form and update coreferential groups, which change in the process of construction of objects due to generation of new objects and attribute values. After this construction terminates, the agents-classes resolve coreferential conflicts by computing and comparing coreferential similarities. The values of coreferential similarities are used further in the processes of lexical and syntactical disambiguation for evaluation of integration of i-objects into a text.

In the nearest future we plan to adopt our approach for reduced forms of anaphora (pronouns, ellipsis) and for other types of coreferential relations such as “set-element”, “part-whole” etc. It will also be useful to extend the definition of semantic similarity to other properties of ontology relations such as intersection, join, closure, inversion and symmetry. Other research direction is the study of dependence of coreferential similarity on text genres such as technical documentation, news, scientific texts, etc. Up to now our approach to semantic text analysis do not take into account homogenous groups in a text. We plan to adopt the approach to consider the case. Now we implement the modules of disambiguation and coreference resolution as the part of our multi-agent system of information extraction for ontology population.

## References

1. Aref, M.M.: A multi-agent system for natural language understanding. In: Conference on Integration of Knowledge Intensive Multi-Agent Systems, p. 36 (2003)
2. Brennan, S.E., Friedman, M.W., Pollard, C.J.: A centering approach to pronouns. In: Proceedings of the 25th Annual Meeting on Association for Computational Linguistics, pp. 155–162. Association for Computational Linguistics, Morristown (1987)
3. Carbonell, J.G., Brown, R.D.: Anaphora resolution: a multi-strategy approach’. In: Proceedings of the 12 International Conference on Computational Linguistics (COLING 1988) (Budapest), pp. 96–101 (1988)
4. Carvalho, A.M.B.R., de Paiva, D.S., Sichman, J.S., da Silva, J.L.T., Wazlawick, R.S., de Lima, V.L.S.: Multi-agent systems for natural language processing. In: Garijo, F.J., Lemaitre, C. (eds.) Multi Agent Systems Models Architecture and Applications, Proceedings of the II Iberoamerican Workshop on D.A.I. and M.A.S, Toledo, Spain, 1–2 October 1998, pp. 61–69 (1998)
5. Elango, P.: Coreference resolution: a survey. Technical report, UW-Madison, 2006, available at: [https://ccc.inaoep.mx/villasen/index\\_archivos/cursoTATII/EntidadesNombradas/Elango-SurveyCoreferenceResolution.pdf](https://ccc.inaoep.mx/villasen/index_archivos/cursoTATII/EntidadesNombradas/Elango-SurveyCoreferenceResolution.pdf)

6. Fum, D., Guida, G., Tasso, C.: A distributed multi-agent architecture for natural language processing. In: Proceedings of the 12th conference on Computational linguistics (COLING 1988), vol. 2, pp. 812–814 (1988)
7. Garanina, N., Sidorova, E., Bodin, E.: A multi-agent text analysis based on ontology of subject domain. In: Voronkov, A., Virbitskaite, I. (eds.) PSI 2014. LNCS, vol. 8974, pp. 102–110. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46823-4\\_9](https://doi.org/10.1007/978-3-662-46823-4_9)
8. Garanina, N.O., Bodin, E.V.: Distributed termination detection by counting agent. In: Proceedings of the 23rd International Workshop on Concurrency, Specification and Programming (CS&P 2014), Chemnitz, Germany, 29 September–01 Oktober 2014. Humboldt-Universität zu Berlin, pp. 69–79 (2014)
9. Garanina, N., Sidorova, E.: Context-dependent lexical and syntactic disambiguation in ontology population. In: Proceedings of the 25th International Workshop on CS&P, pp. 101–112. Humboldt-Universität zu Berlin (2016)
10. Grosz, B.J., Weinstein, S., Joshi, A.K.: Centering: a framework for modeling the local coherence of discourse. *Comput. Linguist.* **21**(2), 203–225 (1995)
11. Harabagiu, S.M., Bunescu, R.C., Maiorano, S.J.: Text and knowledge mining for coreference resolution. In: Proceedings of the Second Meeting of the North American Chapter of the Association for Computational Linguistics on Language Technologies, pp. 1–8. Association for Computational Linguistics (2001)
12. Hobbs, J.: Resolving pronoun references. In: Grosz, B.J., Sparck Jones, K., Weber, B.L. (eds.) *Readings in Natural Language Processing*, pp. 339–352. Morgan Kaufmann Publishers Inc., Los Altos (1986)
13. Kibrik, A.A.: Anaphora in Russian narrative discourse: a cognitive calculative account. In: Fox, B. (ed.) *Studies in Anaphora*, pp. 255–304. John Benjamins, Amsterdam (1996)
14. Kibrik, A.A., Dobrov, G.B., Khudyakova, M.V., Loukachevitch, N.V., Pechenyj, A.: A corpus-based study of referential choice: multiplicity of factors and machine learning techniques. In: *Text Processing and Cognitive Technologies, Cognitive Modeling in Linguistics: Proceedings of the 13th International Conference*, Corfu, pp. 118–126 (2013)
15. Kononenko, I.S., Sidorova, E.A.: Language resources in ontology-driven information systems. In: *First Russia and Pacific Conference on Computer Technology and Applications*, 6–9 September 2010, Vladivostok, Russia, pp. 18–23 (2010)
16. Mitkov, R.: Anaphora resolution: the state of the art (1999). <https://pdfs.semanticscholar.org/e782/00b1e3ba2a72de1ca9b9b2c5efa775151bfa.pdf>
17. Mitkov, R.: Anaphora resolution. In: Mitkov, R. (ed.) *The Oxford Handbook of Computational Linguistics*, pp. 266–283. Oxford University Press, Oxford (2003)
18. Petasis, G., Karkaletsis, V., Paliouras, G., Krithara, A., Zavitsanos, E.: Ontology population and enrichment: state of the art. In: Paliouras, G., Spyropoulos, C.D., Tsatsaronis, G. (eds.) *Knowledge-Driven Multimedia Information Extraction and Ontology Evolution*. LNCS (LNAI), vol. 6050, pp. 134–166. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20795-2\\_6](https://doi.org/10.1007/978-3-642-20795-2_6)
19. Prokofyev, R., Tonon, A., Luggen, M., Vouilloz, L., Difallah, D.E., Cudré-Mauroux, P.: SANAPHOR: ontology-based coreference resolution. In: Arenas, M., et al. (eds.) *ISWC 2015*. LNCS, vol. 9366, pp. 458–473. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-25007-6\\_27](https://doi.org/10.1007/978-3-319-25007-6_27)
20. Rich, E., LuperFoy, S.: An architecture for anaphora resolution. In: *Proceedings of the Second Conference on Applied Natural Language Processing (ANLP-2)*, Texas, USA, pp. 18–24 (1988)

21. Sidorova, E.A., Kononenko, I.S.: Representation and use of the genre structure of documentation in text processing. In: Proceedings of the Science-Intensive Software (SIS-09) - PSI 2009 Satellite Workshop, Novosibirsk, Russia, June 2009, pp. 248–254. Siberian Science Publisher, Novosibirsk (2009). In Russian
22. Soon, W.M., Ng, H.T., Lim, D.C.Y.: A machine learning approach to coreference resolution of noun phrases. *Comput. Linguist.* **27**(4), 521–544 (2001)
23. Wilks, Y.: Preference semantics. In: Keenan, E. (ed.) *The Formal Semantics of Natural Language*. Cambridge University Press, Cambridge (1975)
24. Zhou, G.D., Su, J.: A high-performance coreference resolution system using a constraint-based multi-agent strategy. In: COLING 2004. [www.aclweb.org/anthology/C/C04/C04-1075.pdf](http://www.aclweb.org/anthology/C/C04/C04-1075.pdf)

# A Framework for Dynamical Construction of Software Components

Efim Grinkrug<sup>1,2(✉)</sup>

<sup>1</sup> National Research University Higher School of Economics, Moscow, Russia  
egrinkrug@hse.ru

<sup>2</sup> Institute for System Programming of the Russian Academy of Sciences,  
Moscow, Russia  
grinkrug@ispras.ru

**Abstract.** A component model enabling to construct new software components from existing ones dynamically, at runtime, without their bytecodes generation is presented with supporting it software framework. The framework is implemented using JavaBeans component model, but is aimed to eliminate its drawback – the inability to create user-defined components without bytecodes generation. To construct user-defined component dynamically, a composed prototype object is built using predefined (hardcoded and/or composed) component instances; that prototype object can provide functionality required and can be transformed at runtime into a new component (instantiable type) whose instances are able to provide the same functionality, but more efficiently. The prototype object is composed using meta-components – the framework provided components to produce user-defined components dynamically.

**Keywords:** Component model · Prototype · Type · JavaBeans  
Components · Properties · Interface · Implementation

## 1 Introduction

Any computer program (from the lowest, hardware point of view) is a composite object consisting of its composing elements (machine instructions). In that sense, computer programming is always component-based [1, 2], at least at the lowest level. Complexity of software development depends on a components set available and on the rules to use them for their composition. A component model is of high importance [3]: it is a set of rules that define what the components are and how they can be reused and/or combined together. Any software architecture is defined in terms of components [4], and any software development technology is inherently component-based and component-oriented: it uses and is aimed to produce components, depending on component model and its implementation framework. That component-based approach is common for many engineering areas including hardware and software development (that similarity was pointed to far in 1968, when notions of component-oriented programming and software engineering were first introduced [5]).

In many component-oriented technologies, components themselves are implemented by means that are very different from those used for utilizing the components later.

For instance, hardware chips are produced using highest technologies while the chips can be combined together on a board manually by soldering.

When a component implementation technology is used also to combine components together (since a component itself is usually a composite entity built using its elementary components), we deal with a higher degree of composition - with integration, that the technology must support. In hardware area it may be illustrated by System on Chip integration (based on correspondent degree of integration available). In software it usually is supported by compiler that can integrate various program constructs during compilation. It means that components integration is performed statically, prior the runtime. There may be more or less dynamic approaches to components integration depending on the goals and components in use (e.g., [6, 7] - to list a few).

Components implementation and components integration technologies may or may not be of the same nature. And they can provide different dynamic abilities and efficiency.

This work is concerned with Java platform, as the most popular one [8], while similar considerations may be applicable for others. We propose a component model and its implementation framework enhancing well known JavaBeans component model [9] to facilitate dynamical components composition – with an ability to build new components at runtime. Initially, JavaBeans component model was positioned by Sun as “the only component model for the Java machine”; since that time many others, like e.g. OSGi component model [7], etc., were introduced, but they expect some set of predefined Java interfaces to be implemented by components, thus putting more static requirements on them. And they usually help to compose dynamically end-user applications, but not their components themselves.

The approach to the component composition was initially described in [10] and reflected the first implementation. In this paper the current state of the framework implementation is presented with more details. This work is focusing rather on dynamical construction of composite components than on other aspects of their life-cycle. We are focusing mainly on the issue: how to use existing components (defined types) instances to create a compound prototype-object that can be transformed dynamically, at runtime, into a new composed component (user-defined type) whose instances provide the same functionality as the source prototype. While discussing that issue, we concentrate on the declarative view on our components (what our components are composed from), rather than on the behavioral view on our components instances.

After this introduction, we describe a motivation for this work and the whole approach of it in Sect. 2; Sect. 3 provides our components definitions and Sect. 4 describes our hardcoded components implementation. In Sect. 5 hardcoded meta-components for prototyping are introduced, and Sect. 6 explains composed component instantiation. After short references to the related works in Sect. 7 we conclude with Sect. 8.

## 2 Motivation

Java-platform is based on object-oriented class-based programming paradigm supported by Java language and Java Virtual Machine (JVM) representing static and dynamic parts of the platform. Java-platform provides support for component oriented programming from the very beginning; the JavaBeans component model was introduced in the first JDK, and still remains popular and widely used.

The JavaBeans Specification [9] describes that component model and defines a JavaBeans component as a serializable Java class having public default (no arguments) constructor. JavaBeans components are expected to be manipulated manually (i.e. dynamically and interactively) in a builder tool, and there are some naming conventions (JavaBeans design patterns) defined for discovering a component features (properties, events, etc.) dynamically, at runtime, using corresponding class introspection (that is based on Java reflection mechanism). A reference implementation of the JavaBeans component framework was provided as JavaBeans Development Kit (BDK) with interactive BeanBox tool to manipulate with the components.

The JavaBeans component, by definition, is a class (possibly along with its supporting classes and other resources), i.e. it is a type (of its' dynamically created instances). It should not be mixed with the component instance (an instance of the type), as it often happens; the difference is valuable for our discussion.

As a class, JavaBeans component can be used as an ordinary (library) class by compiler that can combine classes statically to produce other classes (and components), as it is defined in supplied sources. Compiler generated classes, compliant with JavaBeans component model, provide new, statically composed, JavaBeans components. In that case we use the same technology (static compilation) for both the composing components production and the resulting composition from them<sup>1</sup>. And we need to have sources defining how to produce that composition.

In contrast, interactive (dynamic) composition approach supported in BeanBox tool deals with components instances. JavaBeans components (types) are loaded into the tool, where they can be manually instantiated. Components instances can be combined together using their properties assignments and/or linked by events passed between them. All that components instances composition is performed in the predefined BeanBox container instance and results in its' composite content. That composite object can work inside the tool and can be dynamically built/edited/customized in parallel with its functioning – provided that components themselves are able to support it and their runtime environment allows for it (as BeanBox does).

But, how can we use that composite object obtained as the result of components instances composition? We can only explore its functionality in that environment and serialize that container instance content (that's why all JavaBeans components must be serializable) for future use in a similar (JavaBeans specification compliant) environment. That serialization/deserialization can be done using various formats

---

<sup>1</sup> Components used as elements of a composition are composing components; a component having been built from composing components is composed component.



(binary, XML, JSON...). And that is how majority of the component-oriented tools work like; especially various GUI-builders, XUL-editors, Scene Builders, etc.

Sometimes, static and dynamic approaches are used together: a component-oriented environment (tool) provides dynamic, interactive means to build/edit/customize a set of interrelated component instances inside its container instance, but uses that process to create the source file under the hood, that then gets compiled (producing a new component statically). That approach, e.g., was used by BeanComposer tool from IBM Visual Age for Java IDE long ago. Some tools put limitations on components instances functionality inside its container, restricting their dynamic abilities to their views usage only (MS Visual Studio Windows Forms Editor works that way).

The two component building scenarios, mentioned above and expected by Java-Beans component model, have disadvantages: they are either static (use compiler as components integration tool) or they are not logically consistent (in our point of view) when deal with dynamics. Indeed, we start with a set of predefined components, i.e. with classes implementing components (types) - we'll call them as base or hardcoded components. Hardcoded components are produced using class-based object-oriented programming paradigm [11] (and by supporting that paradigm tools and environment of Java-platform). But, having been loaded into a builder tool, predefined components are not expected to produce there a new composed component dynamically: the components can only produce a content of (the tool predefined) container instance, consisting of the used components instances combined together. We can consider that composite instance as a prototype object with functionality we have expected; but, the only way to use it – is copying it through serialization/deserialization (in some form). Thus, having started from class-based object-oriented paradigm we have resulted in prototype-based paradigm usage - that looks like a contradiction (in our point of view), or a kind of ideological downshifting.

Some analogies may be found in other engineering areas. There may be two approaches to create an electronic device. First is – to develop its principal scheme (i.e. a type) based on experience in that area and using some hardware definition language (HDL), e.g., as in [12], then – to produce the device instance(s) according the scheme. Another is – to build a prototype of the device (step by step), extract the scheme from that working prototype and then use that scheme in production. The latter case reminds casting molds usage: first, a prototype is created from some flexible and tunable material; and then the prototype features are extracted and fixed into the immutable mold (type), used in mass production (of the type instances).

A prototype of a type should have more “degrees of freedom” than an instance of the type that is created from the prototype. Instances of components produced with context independent assumptions should be tunable according their concrete usage context requirements. It is desirable for a prototype object that is built from the components instances, to be customizable by means of its elements properties setting, and to have then some of the properties marked as immutable, having their values defined during the prototype elements customization for concrete usage context(s).

For example, some Lego components are produced without knowledge about their future use in specific constructions, and their connection pins remain visible when they will not be used for connection in a given special case. It reduces the final quality of the construction which does not need that extra degree of freedom.

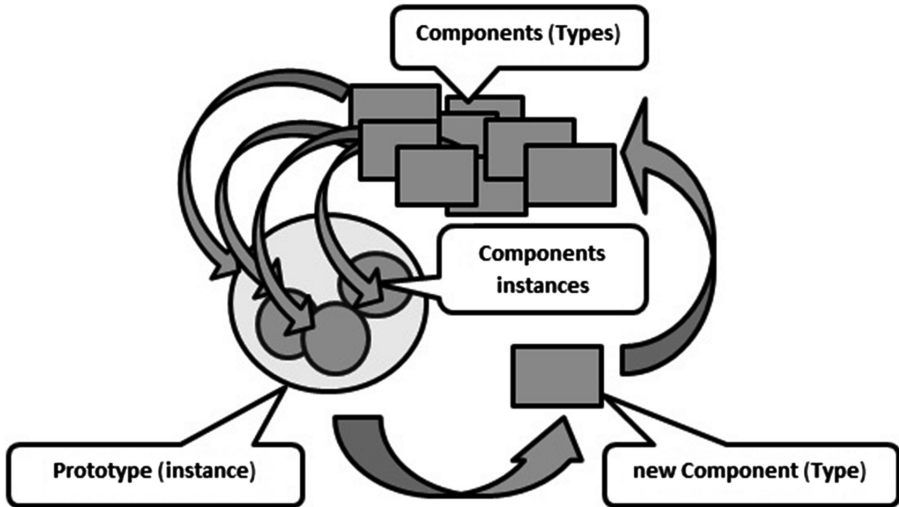


Fig. 1. Creating composed component using its prototype to type transformation

Below we discuss the component model and its implementation framework that support dynamical components composition by means of prototype instance construction from component instances and dynamic prototype to type transformation, as illustrated at Fig. 1.

### 3 Component Interface and Implementation

Our goal is to build new components (composed types) dynamically, i.e. without bytecode generation (by compiler or by other means). In Java-platform it is impossible to have new types (classes) defined dynamically without having their bytecodes loaded by a `ClassLoader`<sup>2</sup>. In order to have our components (types) definable at runtime without having their bytecodes generated, we need to have some extended type system, built using Java-platform (i.e. above it) and extending it to support dynamical type definitions. We call that superstructure BeanVM, since we try to extend the type system of the JavaVM and do it using JavaBeans components.

Our component is a type instantiable without arguments passing and represented by a triplet:  $type = \{name, interface, implementation\}$ . The name uniquely identifies the type, interface represents the type for its users, and implementation part defines how the type is implemented. Our components (types) can be implemented in two forms: hardcoded (having them loaded by JavaVM `ClassLoader` from their bytecodes), and composed (having them defined dynamically without having their classes generated). Both hardcoded and composed types (components) are registered in BeanVM type

<sup>2</sup> Java platform can deal with specific “synthetic” classes defined dynamically by JVM itself for specific purposes.

system, using their names and BeanVM TypeLoader hierarchy support. Hardcoded components represent base (predefined) components, while composed components represent user-defined components in our component model.

We expect to operate with components instances in a manner that is independent on their types implementation – using the interface of their type and the means in it, that are equally applicable both for hardcoded and composed components. To support that, we state that the only way to interact with our component instances is their properties access that we support with correspondent Java API (named as BeanAPI). Hence, interface of any type is defined by its set of the property types, with the property type defined as follows:

$$propertyType = \{name, valueType, access, value\}, \text{ where:}$$

- *name* is the name of the property in the interface;
- *valueType* is a type of the property value in BeanVM type system;
- *access* is a list of operations that are allowed for the property (all supported access rights include Write (W), Read (R), Bind (B) and – for indexed properties only – indexed Write (w) and indexed Read (r));
- *value* is the property default value a component instance has upon instantiation.

Functionality and behavior of component instances are defined in their type definitions as reactions on their property value changes. Detailed description of the mechanisms involved in reactive component instances behavior is beyond the scope of this paper, focusing mainly on declarative aspects of the component model.

Our hardcoded components are specific JavaBeans components (by construction). That allows for manipulations with them in existing JavaBeans compliant tools, and demonstrates our component model as an extension of the JavaBeans component model. But our hardcoded components are implemented in Java as (direct or indirect) subclasses of their common Bean superclass that provides support for Java classes to BeanVM types mapping (dynamically) and BeanAPI implementation. To have an arbitrary JavaBeans component importable in our environment we provide our hardcoded component – BeanAdapter, whose instances wrap JavaBeans component instances and export our BeanVM type interface from them.

BeanAPI provides methods to set a property value having write access permitted for it, to get a property value having its read access permitted, and to bind a property value change listener with a property having that bind access permitted. As for JavaBeans components, we distinguish scalar and indexed property types supporting indexed accessors (and their usage permissions) as well.

We expect to create and use new types at runtime, including their use as property value types for types to be generated in the future, etc., and we do not expect to perform code generation for our composed types (at least for now<sup>3</sup>). We expect to have the same

---

<sup>3</sup> We can use bytecode generation as a kind of just in time compilation, similar to JIT support in JVM [13], as an optimization technique. In that case, component production technology will be the same as component composition technology, producing components with “higher degree of integration” on the fly. But, first, we need to have “source information”.

mechanism to access property values for both hardcoded and composed component instances. When implementing BeanAPI and hardcoded components instances with their properties, we represent property values as objects (of `java.lang.Object` class), and control property assignment dynamically, at runtime.

Implementation part of hardcoded component is represented by its implementation class. Its internal structure is not visible from BeanVM point of view (it looks as atomic entity from BeanVM perspective). Implementation part of composed component (type) consists of the composing types, describing how component types should be used to implement the composed type instances. Next sections describe how hardcoded and composed types are created, instantiated and used at runtime.

## 4 Hardcoded Component Implementation

To define a hardcoded component we have to compile its implementation class having it defined and compiled using our component model implementation framework. The name of the implementation class will be the name of the hardcoded BeanVM type implemented by that class. Our hardcoded components are JavaBeans components inherited from the framework supplied base class `Bean` (i.e. they are our framework specific JavaBeans). Hardcoded component properties should be declared according JavaBeans design pattern for property declaration, but their accessor methods must be encoded using our BeanAPI. Below we describe the ideas behind that.

Any instance of our component (for hardcoded components – a `JavaBean` instance in JavaVM perspective) is a wrapper of its own internal implementation provided by the instance of the BeanVM type (in BeanVM perspective). For hardcoded components that correspondence is established dynamically in context of the constructor of the `Bean` superclass (that gets inevitably executed during hardcoded component instantiation). In case of the very first instantiation of hardcoded component its' BeanVM type is first created and registered. That class to type mapping is accomplished using the static method `Type.forClass(Class c)` of the framework supplied abstract class `Type` returning the type object created from the given loaded class. In fact, the factory returns the type object as an instance of the concrete subclass of the `Type.class`, - so the following invariant always holds true:

```
type.getType() == Type.forClass(Type.class).
```

BeanVM type of the `Type.class` is the type of all BeanVM types, like `Class.class` in JavaVM is the class of all JavaVM classes. If `c` is our hardcoded component implementation class, then the following is true:

```
c.newInstance().getType() == Type.forClass(c).
```

Hardcoded component (BeanVM type) is a wrapper of its implementation class, while that implementation class instance (an object of the class) is a wrapper of an instance of that BeanVM type.

The primitive `Type.forClass(Class c)` provides a hardcoded type for any Java class and uses java class introspection mechanism to extract property descriptors, as defined by JavaBeans specification [9]. BeanVM type system distinguishes hardcoded types created for our hardcoded components (i.e. from classes inherited from our Bean class), for java arrays classes, for other JavaBeans classes, and for all other Java classes (that kinds of hardcoded types are implemented by corresponding subclasses of the abstract `Type.class`). For a given class, all types for all its superclasses are created recursively, and all types for all classes that are used in the given class property descriptors (as its property value type classes) are created as well. The property descriptors and property value types (that are created from property value classes) are used to create property types that define the hardcoded type interface in BeanVM. Additionally, for hardcoded components only, during the type creation procedure, their initial instance is created and initialized providing default values for its property types. For java array classes, type creation procedure performs class to type mapping for the array element class as well. For JavaBeans classes that are not our components their adapter types are created.

The Bean superclass provides BeanAPI - a set of methods to operate with properties of an instance according property access permitted and defined in the property type (with dynamic access control). All values for property setting in BeanAPI are passed as objects, but are checked dynamically against the property value type. To implement property accessors defined in a component class according JavaBeans design pattern, hardcoded component author delegates them to the instance (of BeanVM type) implementation using BeanAPI; when using BeanAPI getters in Java explicit cast to the property value class may be needed. Behavior for a specific hardcoded component is implemented using protected overrides of methods reacting on property value changes.

One of the important issues of any component-based technology is connecting a composed component external interface with the component internal implementation. We define an interface of a (hardcoded and/or composed) component as a set of property types. We need to describe how properties of an interface of a composed component are defined without source code compiling (dynamically), how they are implemented later in composed component instances and how they are connected with properties of the components instances in internal implementation. Next section describes how it is done in component-oriented manner in our framework and how it is related with hardcoded component implementation details – depending on the component instance usage context.

## 5 Prototype Construction

To define new composed component dynamically, we need to define its type name, interface and implementation.

Type names in BeanVM are similar to class names in JavaVM, and used by TypeLoaders that participate in type creation, loading and registration at runtime. The primitive `Type.forName(String typeName)` can load already defined types from class (using `Type.forClass(Class.forName(String typeName))`) or

from other component description source by its name (using a corresponding loader/parser), depending on what will be met first.

To define interface and implementation for new composed component dynamically we use a two-step procedure (as was illustrated at Fig. 1): first, we create a prototype object, and, second, we transform that prototype into the new type using primitive `Type.fromPrototype(Prototype p)`. The prototype object here is a composite object that is to be built dynamically using special hardcoded components instances (among others) that are provided specifically for the prototype composition and prototype to type transformation (i.e., using meta-components to define components).

## 5.1 Prototype Interface Construction

The Prototype meta-component (hardcoded type) has three interface properties: “name”, “interface” and “implementation” having corresponding value types `String`, `PrototypeInterface` and `PrototypeImplementation`<sup>4</sup>.

When the prototype instance will be dynamically transformed into the new BeanVM type, the value of the prototype property “name” will become the name of that type.

The property “interface” of the prototype instance should contain a `PrototypeInterface` instance that is used to define the prototype properties dynamically. To support that, `PrototypeInterface` meta-component has two indexed property types with names “variables” and “properties”. The value type of the property “variables” is the array type of typed variables (`TypedVariable[]`), with all possible access rights (i.e., permitting all possible operations on indexed property: Write, Read, indexed Write, indexed Read, and Bind to register property change event listeners) provided by design. The value type of the property “properties” is the array type of Property-objects (`Property[]`) without Write and indexed Write access from outside. It means that values for the property “properties” are set internally by `PrototypeInterface` meta-component behavior - as the reactions on the property “variables” values changes. When a set of typed variables is changed, the new set of properties, that are produced based on that variables, is generated and made available as an indexed property “properties” value.

Typed variables are our components as well. In contrast with hardcoded components inherited from our Bean class (and having a set of properties), they all have the only property named “value” with the value type defined at typed variable type creation. It means that the type to be instantiated in order to create a typed variable instance is defined in BeanVM “synthetically”: type of all typed variable instances having the given value type is derived from that value type and registered in BeanVM type system. We provide both scalar and indexed typed variable types. All typed variables are writable, readable and bindable (i.e., can be bound using their value change listeners), with indexed variables additionally supporting indexed read and write access. `TypedVariable` and `IndexedTypedVariable` types, as our components, are instantiable without arguments and can create their instances to store values of types compatible with value types they were derived from. The value type of the typed

---

<sup>4</sup> We omit package names from type names for short.

variable is used for dynamic type control during assignments. The BeanVM type system maintain the (extendable) register of types that are not components, but can be used as value types having specific (not null) defaults; that default values are used for typed variable instances initialization. The default value for all typed variables having our component as their value type is null.

Typed variable instances form a typed “sensitive” (i.e., potentially listenable) memory of our component instances for implementing properties in prototypes. They represent “pins” to connect different component interface properties together when constructing a composite prototype object.

Property objects are not our component instances, but they are (a kind of) contexts of typed variable component instances. Properties cannot be created in context-less manner, in contrast with components, but they represent usage contexts of typed variable instances. That is how `PrototypeInterface` component instance works: it creates and exports these contexts for a given set of typed variable instances. Note that typed variable components instances (typed variables) can be used (shared) by many contexts. That sharing is used to define how an interface of a composed component is connected with its implementation components (when the prototype has been transformed into the corresponding type).

Our framework provides contexts for any our component instance; contexts are used in BeanVM component instances addressing mechanism. That context objects connect the component instance with its users that operate with the component instance via its contexts (or get notified about the component instance property change). Property object – as a special case of typed variable instance context – has its own property “access” to set access rights restriction (this context specific) for the typed variable it refers to. Any access restriction defines access rights that are less or equal comparing with that of referenced typed variable instance. When a property is created in `PrototypeInterface` instance (as a context of some typed variable), it is implemented (by the `TypedVariable` contexts factory) as a `PrototypeProperty` object, having additionally its property “name” - to assign the property with user defined name instead of a name generated by default (property names are checked to be unique in the given `PrototypeInterface` meta-component instance).

## 5.2 Prototype Implementation Construction

If some specific functionality (beyond storing a set of typed values in prototype properties) is expected, a content of `PrototypeImplementation` hardcoded meta-component instance should be constructed from available component instances, and some of them should interact with the prototype interface elements. These composing component instances in a prototype implementation will be transformed into *composing types* – elements of the resulting composed type definition produced from the prototype.

Hardcoded `PrototypeImplementation` meta-component has indexed property “components”, where an array of our component instances can be set and/or modified. All components are instantiated without arguments passing, then their property values can be assigned and edited using their value assignments, thus forming a directed graph of referred component instances (since various component instances can refer to the

same instance having it as their properties value). To have that directed graph traversable without falling into an infinite loop, we state that the graph must be an Directed Acyclic Graph (DAG); the correspondent property value assignment control is performed preventing from cyclic references. That graph defines declarative aspect of its container content; behavioral aspect of component instances container is defined by events propagation graph that defines how property change events generated in component instances are used to set new property values in others (i.e. event routes graph). Property change events are used in event routes to get newly generated source property value from an event received from one component instance and to set that value into a destination property of another component instance (provided the properties are value type compatible).

In principle, it is possible to implement interaction between a prototype interface properties with properties of some implementation component instances using event routes. It might be the only way in case we cannot influence on component instances implementation (i.e., considering them as “black boxes”). That way is based on passing information by value. In our component model we can influence on our component instances implementations having them inherited from their common Bean superclass. We can consider that component instances as “grey boxes” at prototyping stage, when defining a prototype implementation.

As it was mentioned in Sect. 2, we expect a prototype object to be more flexible and tunable than any instance of the type the prototype is the source for. That’s why we distinguish two variants of an instance implementation for our components in their Bean superclass: *prototype-based* implementation and *type-based* implementation. The prototype-based component instance implementation is used when the instance is being created for prototyping purposes and needs to be tunable according its usage context under construction. The type-based component instance implementation is used when the type of the instance being created is fully defined and refined for its specific usage context. It is used when we instantiate components participating in the composed type definition (to create its composite instance). That context dependent instance construction is illustrated by the code snippet below:

```
public class Bean ... {
    private final BeanAPI thisInstance;
    public Bean () {
        thisInstance = BeanVM.implementBean(this);
    }
    ...
}
```

Any component instance, implemented as a subclass of our Bean superclass, at the instance creation time gets its own internal implementation produced by BeanVM factory called by Bean class default constructor. That factory knows what the instance under creation is being created for (it knows the instantiation context). By default, all instances get their prototype-based implementation for tuning them as needed.

In prototype-based implementation an instance of component (i.e. an instance of already existing type) is implemented as a set of variables (with each variable



corresponding to a property type that is defined in the existing type interface) and these variables contexts are child contexts of the root context provided by that prototype-based instance implementation object.

These variables contexts are implemented as `InstanceProperty` objects, that are similar with `PrototypeProperty` objects (Sect. 5.1), but, in contrast with `PrototypeProperty`, they have their property types already defined. That defined property types already contain property names, and are used when performing dynamic value type control (against the value type from the property type). Instance properties cannot be renamed, but can support context related access restriction (providing no more access rights than in property type). Both `PrototypeProperty` and `InstanceProperty` are implemented as subclasses of an abstract `Property` class (that, in its turn, is a kind of variable context).

Having properties in prototype-based instance implementation defined as a set of contexts for variables, we can make that property set supporting the variables sharing. That sharing can be performed by our special hardcoded meta-component – `ComposingPrototypeInterface` (its name was intended to highlight its goal as a tool that uses a created component instance as a prototype for composing type when prototype interface connection is needed).

The `ComposingPrototypeInterface` instance is reusable. It gets a component instance in its property “bean” and exports that component instance properties through its own indexed property “properties” having value type `Property[]`. When a new property value then is assigned to the “properties” element at a given index, then the variable of the given property is extracted and its new context as a given component instance property is created (provided all relevant validation checks are successfully performed). If we want to use (by sharing) the prototype interface property variable as the existing component instance property variable, we perform the following scenario:

- create a `PrototypeInterface` instance with its `PrototypeProperty` objects, and extract its property to be linked with an implementation component instance property;
- create a component instance with a property to be linked with the interface property extracted above, any property index can be extracted from the component interface using `getPropertyIndex(String name)`- method;
- create an instance of `ComposingPrototypeInterface` and assign its property “bean” with a component instance as its value; the component instance properties become visible as “properties” value;
- use indexed setter for the property “properties” to assign the extracted interface property as a value using the index of a property found above; the variable of the prototype property will be reused (shared) as a variable of component instance property using its’ newly created context (instance property).

That scenario is successful only when all relevant dynamic checks, including value type and cyclic references control, succeeded; otherwise exception occurs.

`ComposingPrototypeInterface` component is a tool that makes the given component instance connectable by reference (by property variable sharing). It enables reuse of a property variable after having made the given component instance observable as a “gray box”, instead of looking like a “black box” otherwise.

### 5.3 Protoinstance Functionality

The `PrototypeInterface`, `PrototypeImplementation` and `ComposingPrototypeInterface` hardcoded meta-components provide builder interfaces that enable to construct a prototype object dynamically from component instances - as composing elements of the container provided by the instance of the `Prototype` meta-component. That `Prototype` container instance can demonstrate its functionality and behavior “in parallel” with its creation process - by making a prototype object immediately workable while it is being built. It means that the object being prototyped can be accessible not only by its builder interfaces, but used through its functionality interface as well.

To communicate with a prototype object using its functionality interface `Protoinstance` meta-component is used. Note that `Prototype` meta-component has three properties only (properties “name”, “interface” and “implementation”, as defined in Sect. 5). `Protoinstance` component provides “an orthogonal” view on the same prototype object that is created using a `Prototype` component; its instance provides the functionality of the container object being built as a prototype, and it has as many properties as it had been defined so far (dynamically) for that prototype object.

Hardcoded meta-components - `Prototype` and `Protoinstance` – are both the two sides of the same coin, where the instance of the first is used to build a compound prototype object, while the instance of the second is aimed to use that prototype object as expected. For any `Prototype` instance `BeanVM` can return corresponding `Protoinstance` object, and vice versa.

The `protoinstance`-object is implemented internally by the corresponding prototype object internal instance representation - by the context tree of the component instances graph (DAG) in the prototype container. When `Protoinstance` component is instantiated, its instance gets the root context of its corresponding instance of the `Prototype` as its internal implementation.

For any instance of hardcoded component that is a component container (i.e., inherited from abstract `ComponentContainer` class) the framework creates and maintains the internal context tree. The root node of the container instance context tree refers to that container instance itself and has no parent context reference. When a component instance is assigned as a value of a property in a node of the component instances graph, the graph is changed, and its’ corresponding context tree is updated. Any change in the graph is reflected in its context tree (that actually represents the graph inside `BeanVM`). Each component instance context refers to (and listens events from) the component instance it was created for. If the type of the component instance defines property types with value types that are components, then the type is responsible for performing the context creation procedure recursively – in depth, so that the context it returns is a parent context of the contexts created for components found in the component instance properties. That context tree maintaining mechanism<sup>5</sup> is used for many purposes:

---

<sup>5</sup> The context tree is different from, e.g. `java.awt.Container/Component` tree that does not support reuse (sharing) of the component instances.

- It helps to control the component graph without its traversing by providing different context-objects for the same (reused) component instances;
- It simplifies DAG validation when properties of a reference types are assigned (context tree helps to check for cyclic references presence);
- It helps for reactivity support since the context tree represents an instance of a container and gets updated according the component graph behavior.

Since our typed variables, used in the prototype interface, are components and the prototype interface properties are represented by their contexts, we can use the root of the prototype container context tree as the internal implementation of the protoinstance object having these properties. That approach enables dynamical prototyping and functioning of the object being prototyped – at the same time.

## 6 Composed Type Instance Implementation

Composed type created from a Prototype instance has its interface and implementation. Interface is created from prototype interface and is defined in terms of property types. Implementation of the composed type is defined in terms of its composing types that are context dependent refinements of the components used in the prototype.

The composing type is represented by a reference to the component (context-less type) used to create a prototype element the given composing type is transformed from during prototype to type transformation, and, additionally, its property types refinements, that can redefine property values and property access rights reflecting these values of the source prototype element at prototype to type transformation time. The composing type represents context dependent usage of the concrete component instance in the prototype of the composed type. And it provides the context dependent information for the type-based component instance implementation.

During composed type instantiation, instances of the composing types are created as instances of the corresponding components, implemented in type-based manner that is more effective, but less flexible, than prototype-based implementation. Specifically, immutable property values are stored (and shared by all instances) in property type refinements of the corresponding composing types, while mutable property values are stored just in the instance property values array – without typed variables usage; the property value type and access control is delegated to the known property type and its refinement, correspondingly.

A composed component instance is a container object having type-based implemented instances of refined components; it provides the same functionality as a protoinstance of the prototype, but benefits from its type-based optimization.

## 7 Related Works

Component-based software engineering is the base of software engineering in general; both terms were introduced at the same time [5]. Since that time a variety of the component definitions and component models they comply with were proposed and

investigated (e.g., [14–16]) for different application areas. This work is focused on dynamic abilities of a component model and its implementation framework applicable for the most popular software development environment – Java platform, in one point of view, and for expected application areas, in other point of view. Detailed overview of the Java related component models is far beyond the scope of this paper. As a related work, the Ptolemy project [17, 18] should be mentioned. In that project declaratively defined composite types are supported with MOML language [18], but their instances are implemented rather by cloning. A sample of another work with related goals is Evolve project [19], aimed to support software evolution. In that project bytecode generation is used to produce new composite components.

Initially, this work was inspired by an attempt to implement VRML and/or X3D standards [20–23] in Java. These standards are inherently component based and provide language constructs for user-defined component descriptions. Despite of the old publications [21, 22] on having that definitions supported in object-oriented style, all known implementations either perform prototype cloning instead of type instantiation or just use the composite descriptions as parameterized macro definitions. The reasons of it were mentioned in Sect. 3, and these standards still remain actual [24].

## 8 Conclusion

The proposed component model and its implementation framework provide a way for dynamical user defined component definitions. Bytecode generation (compiling) is not needed: new components (composite types) can be produced dynamically from working prototype. When components are used in known contexts having been prototyped first, time and space optimizations are available for instances implementation.

Bytecode generation still remains possible; in BeanVM point of view it can play the role similar to that of JIT-compiler for JavaVM. Components defined dynamically can provide the source information for it.

The framework proposed is implemented based on well-known JavaBeans component model, and interactive component manipulating environment (enhancing SDK) is to be developed. That tool will be able support dynamic component programming without violating initial object-oriented class-based programming paradigm.

As an immediate application of the framework the object-oriented enhancements of the X3D prototyping with corresponding tool support are expected, while the framework can be used for a variety of other component-oriented applications.

## References

1. Sommerville, I.: Software Engineering, 10th edn. Pearson, New York (2015)
2. Wang, A.J.A., Qian, K.: Component-Oriented Programming. Wiley, New York (2005)
3. Lau, K.-K., Wang, Z.: A Survey of Software Component Models (second edition), School of Computer Science, The University of Manchester, Preprint Series, CSPP-38 (2006)

4. Paul, C., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Stafford, R.N.J.: Documenting Software Architectures: Views and Beyond, 2nd edn. Addison-Wesley, Boston (2010)
5. McIlroy, D.: Mass-produced “software components. In: Naur, P. Randell, B.: Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968”. Scientific Affairs Division, NATO, Brussels
6. Spring Framework. <https://spring.io/projects>
7. OSGi Technology. <https://www.osgi.org/developer/architecture/>
8. Tiobe index. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>
9. JavaBeans API Specification. <http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html>
10. Grinkrug, E.: Dynamic component composition. *Int. J. Softw. Eng. Appl.* **5**(4), 84–101 (2014)
11. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer, New York (1996). <https://doi.org/10.1007/978-1-4419-8598-9>
12. Nusan, N., Schoken, S.: The Elements of Computing Systems: Building a Modern Computer From First Principles. The MIT Press, Cambridge (2005)
13. Oaks, S.: Java Performance: The Definitive Guide. O’Reilly Media, Sebastopol (2014)
14. Meyer, B.: The grand challenge of trusted components. In: Proceedings of ICSE 2003, pp. 660–667. IEEE (2003)
15. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Boston (2002)
16. Heineman, G., Councill, W. (eds.): Component-Based Software Engineering: Putting the Pieces Together, 2nd edn. Addison-Wesley, Boston (2001)
17. The Ptolemy Project. <http://ptolemy.eecs.berkeley.edu/index.html>
18. Lee, E.A., Neuendorffer, S.: Technical Memorandum UCB/ERL M00/12, Dept. EECS. <http://ptolemy.eecs.berkeley.edu/publications/papers/00/moml/>
19. McVeigh, A.: Creating, Reusing and Executing Components in Evolve. <http://www.intrinsarc.com/evolve>
20. The Virtual Reality Modeling Language. ISO/IEC 14772. [www.web3d.org](http://www.web3d.org)
21. Beeson, C.: An object-oriented approach to VRML development. In: Proceedings of the Second Symposium on Virtual reality Modeling Language, VRML 1997, pp. 17–24 (1997)
22. Diehl, S.: VRML ++: A language for object-oriented virtual reality models. In: Proceedings of the 24th International Conference on Technology of Object-Oriented Languages and Systems, TOOLS, Beijing, pp. 141–150 (1997)
23. Brutzman, D., Daly, L.: X3D: Extensible 3D Graphics for Web Authors. Elsevier, Amsterdam (2007)
24. The Instant Reality Framework. <http://www.instantreality.org/>

# A Transformation-Based Approach to Developing High-Performance GPU Programs

Bastian Hagedorn<sup>1</sup>(✉), Michel Steuer<sup>2</sup>, and Sergei Gorlatch<sup>1</sup>

<sup>1</sup> University of Münster, Münster, Germany  
{b.hagedorn,gorlatch}@wwu.de

<sup>2</sup> University of Glasgow, Glasgow, UK  
michel.steuer@glasgow.ac.uk

**Abstract.** We advocate the use of formal patterns and transformations for programming modern many-core processors like Graphics Processing Units (GPU), as an alternative to the currently used low-level, *ad hoc* programming approaches like CUDA or OpenCL. Our new contribution is introducing an intermediate level of *low-level patterns* in order to bridge the abstraction gap between popular high-level patterns (*map, fold/reduce, zip*, etc.) and imperative, executable code for many-cores. We define our low-level patterns based on the OpenCL programming model which is portable across parallel architectures of different vendors, and we introduce semantics-preserving rewrite rules that transform programs with high-level patterns into programs with low-level patterns, from which executable OpenCL programs are automatically generated. We show that program design decisions and optimizations, which are usually applied *ad-hoc* by experts, are systematically expressed in our approach as provably-correct transformations for high- and low-level patterns. We evaluate our approach by systematically deriving several differently optimized OpenCL implementations of parallel reduction that achieve performance competitive with OpenCL programs which are manually written and highly tuned by performance experts.

**Keywords:** Parallel programming · Rewrite rules  
Algorithmic patterns · GPU · OpenCL · Code generation  
Skeletons · Transformations

## 1 Motivation and Related Work

Although systems with many-core accelerators, like Graphics Processing Units (GPUs) and Intel Xeon Phi are an inherent part of modern high-performance computing, achieving high application performance on these systems remains a challenging task even for experienced programmers. Usually, an initial, intuitively correct version of an application is iteratively improved by applying *ad-hoc* optimizations using experience-motivated “rules of thumb”. Writing high-performance code for many-core architectures requires explicit management of

available resources which nowadays comprise of: (1) a hierarchy of several hundreds or even thousands of processing units (cores) able to execute multiple concurrent threads which are additionally organized in groups, warps, etc., and (2) a memory hierarchy consisting of multiple cache levels, software-managed local memory for groups of threads, and global memory. Therefore, high-performance code is usually written by experts using low-level programming approaches like OpenCL [10] or CUDA [8] which require the programmer to explicitly manage both thread and memory hierarchies.

The challenges of the state-of-the-art GPU programming are demonstrated in the recent GPU programming guide [8], where performance experts of Nvidia Corp. consider an allegedly very simple application – the reduction of an array (e. g., the summation of array elements). In order to achieve high performance on GPUs, they develop seven differently optimized implementations for this simple example. The eventually achieved speedup is up to 30 times compared to the initial version, which on the one hand emphasizes the importance of program optimizations for GPUs, and on the other hand demonstrates how difficult and hardware-specific the optimization process is, even for simple applications.

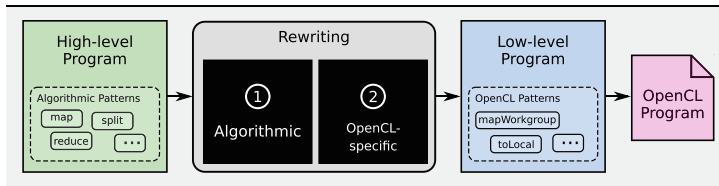
In this paper, we propose a systematic approach to program development for systems with GPUs. A program is expressed using high-level algorithmic patterns and then systematically transformed into a program with novel low-level patterns, from which high-performance GPU code is automatically generated.

Our approach is inspired by early work on transformational programming [3–5]. We improve the state of the art by formally introducing low-level, OpenCL-specific patterns and formalizing the device- and application-specific transformations for parallel GPU programs, which before were only informally described in optimization guides of hardware vendors like [8]. The Lift framework [13,16] provides a prototype implementation of our low-level patterns and rewrite rules, and it demonstrates that they work well for a broad class of real-world applications. There exist libraries based on the concept of algorithmic patterns (skeletons [7]): SkelCL [15], MUESLI [11] or FastFlow [1]. While they rely on hard-coded and hardware-specific implementations with a fixed set of optimizations and are therefore inherently not performance-portable, our approach allows to systematically derive optimizations by rewrite rules which enable applying different optimizations for different devices. Functional approaches like Accelerate [6], Harlan [9] and Obsidian [17] rely either on predefined implementations or on time-consuming code analysis. In contrast, our approach expresses hardware-relevant paradigms of OpenCL in functional code which allows us to express low-level optimizations using rewrite rules.

In this paper, we make the following main contributions: in Sects. 2 and 3, we introduce a novel transformation process to systematically develop programs for parallel systems using high-level patterns, and in Sect. 4, we formalize low-level optimizations as rewrite rules allowing the systematic derivation of optimized programs from a high-level program. In Sect. 5, we experimentally evaluate programs generated by our approach: and we demonstrate their performance in comparison to high-performance libraries like Nvidia cuBLAS [12].

## 2 Our Approach: Patterns and Transformations

We use well-known high-level patterns like *map* and *reduce*, also known as algorithmic skeletons [7], to express applications as computations on (multi-dimensional) arrays. Although these patterns themselves are simple, they can specify many real-world applications like N-body-simulations [16] or medical imaging [14]. In our approach, the transformation process systematically rewrites a high-level pattern-based program describing *what* to compute into a low-level pattern-based program describing *how* the computation is organized within the OpenCL programming model. The resulting low-level program is used to generate executable OpenCL code. We use OpenCL as our low-level programming model since it is portable on a broad variety of architectures including GPUs, multicore CPUs, Intel Xeon Phi, and FPGAs.



**Fig. 1.** Transformation approach: a high-level, pattern-based program is systematically transformed into a low-level program in two phases: 1. Algorithmic Rewriting: the high-level program is decomposed in sub-parts that can be executed in parallel. 2. OpenCL-specific Rewriting: the decomposed program is transformed and optimized using OpenCL-specific patterns, from which OpenCL code is generated.

Figure 1 shows how a high-level program is transformed to high-performance OpenCL code in a two-phase transformation process:

1. *Algorithmic Rewriting.* In the first phase, we rewrite the algorithmic structure of a high-level program: we decompose it into parts which can be executed in parallel. For example, instead of reducing an array in one step, we derive a program that expresses the reduction as a tree-based reduction (processing parts in parallel) followed by a final reduction as suggested by the Nvidia experts in [8].
2. *OpenCL-Specific Rewriting.* In the second phase, we transform the decomposed high-level program to a program with low-level, OpenCL-specific patterns by means of rewrite rules that map high-level patterns to the OpenCL's thread hierarchy and data to the OpenCL's memory hierarchy. Efficient OpenCL code can be automatically generated from the resulting low-level program.

## 3 Algorithmic Patterns and Rewriting

Our high-level patterns are similar to those used in functional programming approaches [3,4], which allows us to reuse already proved rewrite rules. Compared to existing languages and formalisms, we use only few selected patterns



for which we can generate high-performance parallel code. This allows us to limit the variety of required rewrite rules while providing an expressive enough language to develop a broad class of high-performance applications.

### 3.1 High-Level Algorithmic Patterns

All our patterns are defined as (higher-order) functions on arrays. An array  $xs$  of length  $n$  containing elements  $x_i$  is denoted as  $[x_0, \dots, x_{n-1}]$ . Higher-dimensional data structures like matrices or cubes are represented as nested multidimensional arrays. Instead of using recursive cons-lists, as used for example in the BMF, we define our patterns on arrays because we target the generation of high-performance OpenCL codes which work on plain C-arrays. We use a notation similar to the BMF and denote function application by a whitespace:  $f x$ . We use the  $\circ$  operator to denote function composition which associates to the right, e. g.,  $(f \circ g) x = f (g x)$ , and has a lower precedence than function application which associates to the left, e. g.,  $f x \circ g y$  is read as  $(f x) \circ (g y)$ .

**Definition 1 (High-Level Algorithmic Patterns).**

$$\mathit{map} f [x_0, \dots, x_{n-1}] = [f x_0, \dots, f x_{n-1}] \quad (1)$$

$$\mathit{reduce} (\oplus) [x_0, \dots, x_{n-1}] = [x_0 \oplus \dots \oplus x_{n-1}] \quad (2)$$

$$\mathit{zip} [x_0, \dots, x_{n-1}] [y_0, \dots, y_{n-1}] = [\langle x_0, y_0 \rangle, \dots, \langle x_{n-1}, y_{n-1} \rangle] \quad (3)$$

$$\mathit{split} m [x_0, \dots, x_{n-1}] = [[x_0, \dots, x_{m-1}], \dots, [x_{n-m-1}, \dots, x_{n-1}]] \quad (4)$$

$$\begin{aligned} \mathit{join} [[x_0, \dots, x_{m-1}], \dots, [x_{n-m-1}, \dots, x_{n-1}]] \\ = [x_0, \dots, x_{m-1}, \dots, x_{n-m-1}, \dots, x_{n-1}] \end{aligned} \quad (5)$$

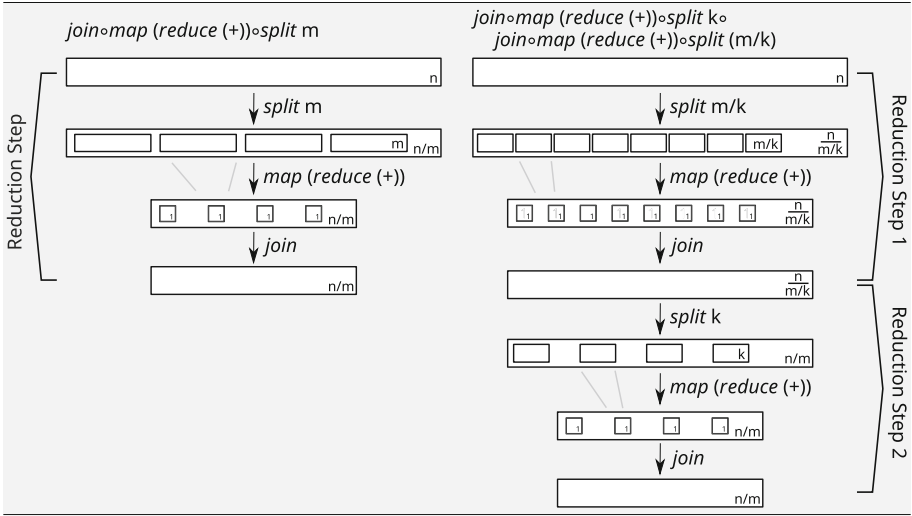
The *map* pattern applies a unary function to all elements of an array. *reduce* combines all elements of an array using an associative binary operator and returns a singleton array. Returning a singleton array instead of a scalar value simplifies the formulation of some rewrite rules. The *zip* pattern combines two arrays of the same length element-wise to an array of pairs. The *split* pattern splits its input into chunks of the specified size  $m$ , i. e., it adds another array dimension. The *join* pattern, also known as *concat*, reduces the dimension of a given array by flattening its two outermost dimensions into one.

### 3.2 Algorithmic Rewrite Rules

In order to systematically transform high-level programs in a semantics-preserving way, we define a set of rewrite rules, also known as algebraic identities, which we denote as  $A = B$ . In an arbitrary program, the left-hand side expression ( $A$ ) of a rule can be replaced with the right-hand side expression ( $B$ ) and vice versa.

For example, the map-distribution rule [4] states that *map* distributes over function composition:

$$\mathit{map} f \circ \mathit{map} g = \mathit{map} (f \circ g) \quad (6)$$



**Fig. 2.** Tree reduction rule: a reduction of  $m$  elements (left) is the same as first reducing  $m/k$  elements and then reducing the temporary results (right).

The *map-promotion* rule [4]:

$$\text{map } f \circ \text{join} = \text{join} \circ \text{map} (\text{map } f) \tag{7}$$

describes handling of two-dimensional arrays: instead of applying  $f$  to the flattened array (produced by *join*) it is also possible to apply (*map f*) on each outer array and join the resulting arrays afterwards. A variation of this rule allows to explicitly introduce an additional array dimension using *split*:

$$\text{map } f = \text{join} \circ \text{map} (\text{map } f) \circ \text{split } m \tag{8}$$

Adding additional dimensions using this rule will become useful when we map computations to the hierarchically structured OpenCL programming model.

Rules can also define relations between more complex compositions of patterns as the following *tree-reduction* rule, provided that  $k$  divides  $m$ :

$$\text{join} \circ \text{map} (\text{reduce } (\oplus)) \circ \text{split } m = \text{join} \circ \text{map} (\text{reduce } (\oplus)) \circ \text{split } k \circ \text{join} \circ \text{map} (\text{reduce } (\oplus)) \circ \text{split } \frac{m}{k} \tag{9}$$

Figure 2 visualizes the tree-reduction rule. The left-hand side shows a single reduction step that consists of: (1) dividing the input into disjoint chunks using *split*; (2) reducing all chunks independently using *map (reduce ⊕)*; (3) combining the results using *join*.

The right-hand side shows two reduction steps, where the first step reduces the array from  $n$  elements to an array of  $\frac{n}{m/k}$ , before the second step reduces the array to  $n/m$  elements, which corresponds to the right-hand side (9). Applying the tree-reduction rule recursively leads to multiple steps that reduce the input

array in a tree-like fashion. Computing reductions as tree-based reductions is one of the optimizations suggested by the Nvidia experts in [8].

A list containing more rewrite rules is given in Appendix A.

*Correctness of Rewrite Rules.* All of the rules in this paper are provably correct with respect to a standard functional denotational semantics of our patterns as defined in [13]. Applying semantics-preserving rewrite rules to pattern-based programs allows us to guarantee the correctness of the derivation process, thus, a derived program always computes the same result as the original.

An example of proving (25) using equational reasoning is given in Appendix B.

### 3.3 Transformation Using Algorithmic Rewrite Rules

Let us consider an example of how the summation of the elements of an array is systematically transformed using rewrite rules starting from the high-level program (HLP): *reduce* (+). We deliberately choose this concise example (which is nevertheless non-trivial for implementing on GPUs as shown in [8]) to discuss our formal approach in depth.

In the following, we use a superscript to denote the composition of the same function, e.g.  $(f \circ f \circ f) = f^3$ . Starting from the high-level program, we systematically perform the following transformations:

$$\begin{aligned}
 & \text{reduce } (+) && \text{(HLP)} \\
 \{ (25) \text{ in Appendix A} \} & \\
 & = \text{reduce } (+) \circ \text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } m \\
 \{ (25) \text{ in Appendix A} \} & \\
 & = \text{reduce } (+) \circ \text{join} \circ \text{map } ( \\
 & \quad \text{reduce } (+) \circ \\
 & \quad \text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } m \\
 & \quad ) \circ \text{split } m \\
 \{ (9) \} & \\
 & = \text{reduce } (+) \circ \text{join} \circ \text{map } ( \\
 & \quad \text{reduce } (+) \circ \\
 & \quad \text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } 2 \circ \\
 & \quad \text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } m/2 \\
 & \quad ) \circ \text{split } m \\
 \{ (9) \} & \\
 & = \text{reduce } (+) \circ \text{join} \circ \text{map } ( \\
 & \quad \text{reduce } (+) \circ \\
 & \quad (\text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } 2)^2 \circ \\
 & \quad \text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } m/4 \\
 & \quad ) \circ \text{split } m \\
 \{ (9) \text{ applied } (\log m) - 2 \text{ times} \} & \\
 & = \text{reduce } (+) \circ \text{join} \circ \text{map } ( \\
 & \quad \text{reduce } (+) \circ (\text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } 2)^{\log m} \\
 & \quad ) \circ \text{split } m && \text{(TP)}
 \end{aligned}$$

Interestingly enough, our transformed program (TP) follows the algorithmic structure of the first version of the parallel reduction described by the Nvidia experts in [8]. The parameter  $m$  is an arbitrary value (as long as it divides the size of the input) used to split the input into chunks of size  $m$ , e.g.,  $m = 128$  as suggested in the Nvidia example. The input is divided by *split* into distinct chunks; then each chunk is iteratively reduced into a single temporary result by pairwise combining and reducing neighboring elements; finally, all temporary results, i.e., the sums of all chunks, are summed up by the leftmost *reduce*.

We continue with the obtained transformed program (TP) for the parallel reduction in the next section and further transform it into a program with OpenCL-specific patterns. The fully optimized low-level program and the OpenCL code generated from it are presented and evaluated at the end of the paper.

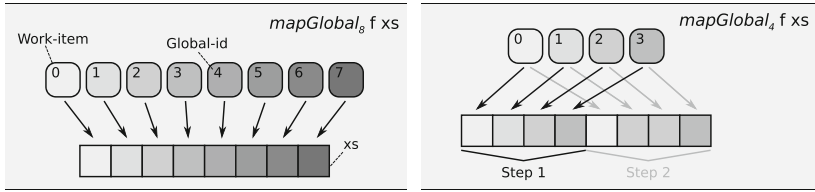
## 4 OpenCL-Specific Patterns and Rewriting

In this section, we introduce OpenCL-specific patterns to specify how programs are mapped onto OpenCL, i.e., how computations are assigned to OpenCL's thread hierarchy, and how data are stored in OpenCL's memory hierarchy. Furthermore, we introduce rewrite rules that transform high-level programs into programs using low-level patterns, which are ultimately transformed to OpenCL.

### 4.1 Exploiting the Thread Hierarchy Using Low-Level Map Patterns

OpenCL [10] is currently a de-facto standard for portable programming of systems with multi- and many-core processors. The OpenCL model differentiates between a *host*, in our case a CPU, and a *device*, e.g., a GPU. The parallel execution of a program, called *kernel*, is performed on the device by multiple threads in the OpenCL's thread hierarchy: threads, called *work-items*, are organized in *work-groups*. Computations within a work-item are performed sequentially. Each work-item has two IDs: a unique *global-id* and a *local-id* which is unique within the work-item's work-group; work-groups have unique IDs themselves. The IDs allow to exploit the thread hierarchy using either only global IDs and therefore work-items directly, or using work-groups and local IDs within them.

Nvidia GPUs add a third level to the thread hierarchy: work-groups are further divided into so-called *warps*, i.e., groups of 32 work-items that are called *lanes* and are executed together in a lock-step manner, i.e., all lanes execute the same instruction at the same time. Although warps and lanes are not captured by the OpenCL 1.2 standard, it is performance-critical to optimize the warp execution for Nvidia GPUs: in particular, since work-items of the same warp are implicitly synchronized, the costly barrier synchronization can be avoided. The current practice of GPU programming requires that low-level, device-specific optimizations are carefully applied by experts with knowledge of the target architecture. We propose low-level, OpenCL-specific patterns and rewrite rules that introduce such optimizations systematically.



**Fig. 3.** Visualization of the *mapGlobal* pattern: The input is divided among available work-items which apply the given function to their assigned elements

We start by introducing several low-level variants of the high-level *map* pattern. Each of these low-level patterns represents a possible realization of the high-level *map* semantics (1) using the different levels of the OpenCL’s thread hierarchy. Our first pattern – *mapGlobal* – specifies how *m* work-items, identified by their global IDs, apply function *f* to all *n* elements of an array in parallel:

$$\text{mapGlobal}_m f [x_0, \dots, x_{n-1}] = [y_0, \dots, y_{n-1}], \text{ where } y_i = f_{(i \bmod m)} x_i \quad (10)$$

Here, we annotate function *f* with a subscript indicating which work-item computes which element: e. g.,  $f_0 x_0$  denotes that the work-item with the global ID = 0 applies function *f* to element  $x_0$ . Definition (10) specializes the definition (1) of the high-level map by prescribing which work-items perform computations. We omit in *mapGlobal* the subscript *m* that specifies the number of work-items, when all available work-items take part in the computation.

Figure 3 shows two possible situations of using the *mapGlobal* pattern. In the simplest case on the left-hand side, the number of work-items equals the size of the input ( $m = n$ ), such that each work-item applies *f* to the input element whose index in the array equals the work-item’s global ID. If there are fewer work-items than input elements ( $m < n$ ), as on the right-hand side of Fig. 3, then all work-items start applying *f* to the *m* leftmost elements in the array and then proceed to the next *m* elements, until *f* is applied to all input elements.

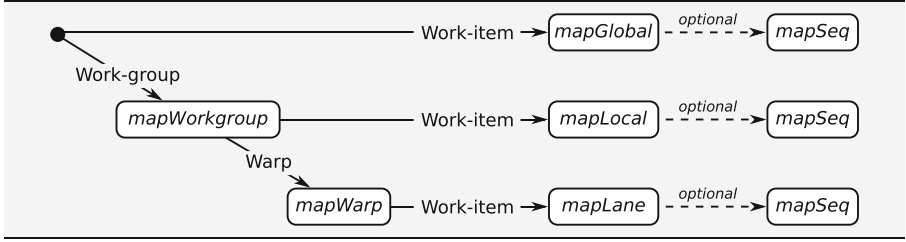
Figure 4 shows the OpenCL pseudo-code which implements *mapGlobal*: a for-loop iterates over the global IDs and applies function *f* to all input elements.

```

mapGlobal f xs
    ↓
for (int g_id = get_global_id(0); g_id < n; g_id += m) {
    output[g_id] = f( xs[g_id] ); }
    
```

**Fig. 4.** OpenCL pseudo-code implementing definition (10) of *mapGlobal*

Our next patterns – *mapWorkgroup* and *mapLocal* – are used to exploit the two-level thread hierarchy of work-groups and work-items in OpenCL. These patterns are defined similarly to *mapGlobal*, with the difference that the subscript



**Fig. 5.** Valid nestings of low-level maps expressing the OpenCL thread hierarchy.

of  $f$  corresponds to the work-group ID and local work-item ID within the work-group, correspondingly. The subscript  $m$  defines the number of work-items in a work-group for  $mapLocal$  and the number of work-groups for  $mapWorkgroup$ . To respect the OpenCL thread hierarchy, the  $mapLocal$  pattern can only occur nested in the  $mapWorkgroup$  pattern. The OpenCL implementation pseudocode for  $mapLocal$  and  $mapWorkgroup$  is almost identical with the code of  $mapGlobal$  in Fig. 4. The only difference is that the call to `get_global_id` is replaced with a corresponding call to obtain the local or work-group ID.

For Nvidia GPUs, we utilize the additional, third level of the thread hierarchy by introducing two patterns,  $mapWarp$  and  $mapLane$ , defined similar to the  $mapGlobal$  pattern. For  $mapWarp$ , the subscript of  $f$  corresponds to the warp ID which is calculated as  $\lfloor local\_id/32 \rfloor$ . In case of  $mapLane$ , the subscript of  $f$  corresponds to the ID of a lane which is calculated as  $(local\_id \bmod 32)$ ; furthermore,  $mapLane$  always has to be nested in a  $mapWarp$ .

Finally, the  $mapSeq$  pattern expresses  $map$  computed sequentially.

Figure 5 provides an overview of the introduced low-level  $map$  variants and visualizes the nestings of them which respect the OpenCL thread hierarchy. To enforce this nesting structure, we introduce a set of rewrite rules to transform nestings of the high-level  $map$  pattern into equivalent low-level expressions:

$$map \rightarrow mapGlobal \quad (11)$$

$$map (map f) \rightarrow mapWorkgroup (mapLocal f) \quad (12)$$

$$| mapGlobal (mapSeq f)$$

$$map (map (map f)) \rightarrow mapWorkgroup (mapWarp (mapLane f)) \quad (13)$$

$$| mapWorkgroup (mapLocal (mapSeq f))$$

$$map (map (map (map f))) \rightarrow mapWorkgroup (mapWarp ( \quad (14)$$

$$mapLane (mapSeq f)))$$

## 4.2 Exploiting the Memory Hierarchy Using Low-Level Patterns

In the following, we introduce a collection of low-level patterns to utilize OpenCL's memory hierarchy consisting of four disjoint memory spaces: *global*, *local*, *private*, and *constant memory*. The global and constant memory are available to all work-items executing a kernel and correspond to the GPU's main memory. The local memory is shared by all work-items of a work-group and corresponds to the fast on-chip memory of a GPU. The private memory is owned by a single work-item and corresponds to the registers of a GPU. Accessing the small private and local memory is several hundred times faster than accessing the larger global memory. Therefore, efficient utilization of the GPU's memory hierarchy is mandatory in order to achieve high performance.

To make the different memory spaces explicit in our low-level programs, we extend the array type with a memory space notation: e. g.,  $[A]_n^{\text{global}}$  denotes the type of an array with  $n$  elements of type  $A$  residing in global memory. The global memory is the default memory space for arrays in OpenCL and input arrays are always allocated in global memory.

We introduce low-level patterns to allow the programmer to change the memory space. In particular, they allow work-items of a work-group to copy data from the global memory to the fast local memory, which is a well-known optimization in OpenCL that can significantly speed up the execution of a kernel. The *toLocal* pattern is defined as follows for an arbitrary memory space  $M$ :  $\text{toLocal} : [A]_n^M \rightarrow [A]_n^{\text{local}}$ . Intuitively, this pattern takes an array located in an arbitrary memory space and returns the same array stored in the local memory. The *toLocal* pattern is, therefore, a hint to our code generator to copy the array into the GPU's local memory space. The *toGlobal* and *toPrivate* patterns are defined analogously, allowing the utilization of the corresponding memory spaces. There is no *toConstant* pattern, because the constant memory is read-only.

For example, consider an array  $xs$  of type  $[[A]_4]_2^{\text{global}}$ . The following program applies  $f$  to all elements using two work-groups with four work-items per group, utilizing local memory and writing the result into the global memory:

```

ys = mapWorkgroup2 (
    toGlobal ◦                               // copy to global memory
    mapLocal4 f ◦                             // apply f in local memory
    toLocal                                   // copy to local memory
) xs

```

The following rewrite rules specify how the low-level memory patterns can be used together with the previously introduced low-level *map* patterns:

$$\begin{aligned} \text{mapWorkgroup } f &\rightarrow \text{mapWorkgroup } (\text{toGlobal} \circ f) & (15) \\ &| \text{mapWorkgroup } (f \circ \text{toLocal}) \end{aligned}$$

$$\begin{aligned} \text{mapLocal } f &\rightarrow \text{toGlobal} \circ \text{mapLocal } f & (16) \\ &| \text{mapLocal } f \circ \text{toPrivate} \end{aligned}$$

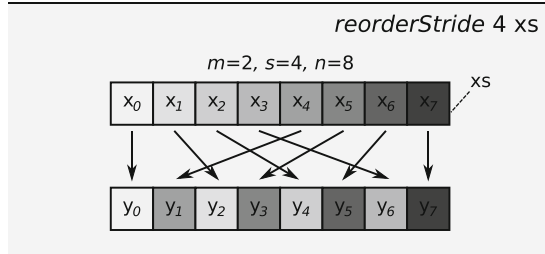
$$\begin{aligned} \text{mapGlobal } f &\rightarrow \text{toGlobal} \circ \text{mapGlobal } f & (17) \\ &| \text{mapGlobal } f \circ \text{toPrivate} \end{aligned}$$

$$\begin{aligned} \text{mapLane } f &\rightarrow \text{toGlobal} \circ \text{mapLane } f & (18) \\ &| \text{mapLane } f \circ \text{toPrivate} \end{aligned}$$

These rules allow individual work-items to use their private memory, and (15) describes the possibility to use the local memory for computations in a workgroup (the use of local memory outside a workgroup is forbidden in OpenCL). To generate a valid OpenCL kernel, the final result of a kernel has to reside in the global memory to be accessible from the host, therefore, a *toLocal* or *toPrivate* has to be followed by a *toGlobal* in a correct low-level program.

As our final low-level pattern we introduce *reorderStride* which enforces a special reordering of arrays in global memory, for  $m = s \times n$ , as follows:

$$\begin{aligned} \text{reorderStride } s [x_0, \dots, x_{n-1}] &= [y_0, \dots, y_{n-1}], \text{ where} \\ y_i &= x_{((i-1) \text{ div } n + s \times ((i-1) \text{ mod } n))} \end{aligned} \tag{19}$$



**Fig. 6.** The *reorderStride* pattern: input elements are reordered using a given stride

Figure 6 visualizes the reordering of an array  $xs$  with 8 elements using a stride of 4. Reordering the elements of an array following (19) ensures that when all work-items access their elements, consecutive work-items access consecutive memory elements at the same time. This access pattern corresponds to so-called *coalesced memory accesses*, which are beneficial on GPUs as multiple memory accesses of work-items can be fused to a single memory access. Here,



$x_2$  is reordered to position  $y_4$ , because  $4 = (2 - 1) \operatorname{div} 2 + 4 \times ((2 - 1) \bmod 2)$ . In the generated OpenCL code, reordering will not be performed by copying the array, but rather by reading the array elements in a different order.

The following rewrite rule introduces *reorderStride* in reductions:

$$\text{reduce } (\oplus) \rightarrow \text{reduce } (\oplus) \circ \text{reorderStride } n \quad (20)$$

We only apply *reorderStride* if the reordered array is reduced afterwards. Therefore, we only change the order in which the elements are combined using  $\oplus$ ; this requires that  $\oplus$  is associative and commutative.

### 4.3 Using Low-Level Patterns to Implement High-Level Reduction

Using the rewrite rules for thread and memory hierarchies, we further derive the parallel program for reduction systematically, by introducing OpenCL-specific low-level patterns. We start with the transformed program (TP) obtained in Sect. 3.3:

$$\begin{aligned} & \text{reduce } (+) \circ \text{join} \circ \text{map } ( \\ & \quad \text{reduce } (+) \circ (\text{join} \circ \text{map } (\text{reduce } (+)) \circ \text{split } 2)^{\log m} ) \circ \text{split } m \\ \{(12)\} \\ & = \text{reduce } (+) \circ \text{join} \circ \text{mapWorkgroup } ( \\ & \quad \text{reduce } (+) \circ \\ & \quad \quad (\text{join} \circ \text{mapLocal } (\text{reduce } (+)) \circ \text{split } 2)^{\log m} ) \circ \text{split } m \\ \{(15) \text{ applied twice}\} \\ & = \text{reduce } (+) \circ \text{join} \circ \text{mapWorkgroup } ( \\ & \quad \text{toGlobal} \circ \text{reduce } (+) \circ \\ & \quad \quad (\text{join} \circ \text{mapLocal } (\text{reduce } (+)) \circ \text{split } 2)^{\log m} \circ \text{toLocal} \quad (\text{LLP1}) \\ & \quad ) \circ \text{split } m \end{aligned}$$

The obtained low-level program LLP1 closely resembles the first optimized OpenCL kernel for parallel reduction as informally described in [8]. In Appendix C we show two more low-level programs, LLP2 and LLP3, also similar to versions from [8]. Like the program LLP1, these are derived from the same high-level program HLP by varying the choice of rewrite rules applied.

### 4.4 Code Generation

Starting with a high-level program consisting of the high-level pattern *reduce*, we systematically derived optimized low-level versions LLP1–LLP3 of the parallel reduction. In this section, we briefly explain how to transform such low-level programs into imperative OpenCL code using our code generator.

```

float sumUp(float x, float y) { return x+y; }
kernel void OCL1(global float* g_idata, global float* g_odata,
                unsigned int N, local float* sdata) {
    local float* sdata1 = sdata;
    local float* sdata2 = &sdata1[128];
    local float* sdata3 = &sdata2[64];
    for (int wg_id = get_group_id(0); wg_id < (N / (128));
        wg_id += get_num_groups(0)) {
        int l_id = get_local_id(0);
        sdata1[l_id] = g_idata[(wg_id * 128) + l_id];
        barrier(CLK_LOCAL_MEM_FENCE);
        int size = 128;
        local float* sin = sdata1;
        local float* sout = ((7 & 1) != 0) ? sdata2 : sdata3;
        for (int j = 0; j < 7; j += 1) {
            int l_id = get_local_id(0);
            if (l_id < size / 2) {
                float acc = 0.0f;
                for(int i = 0; i < 2; ++i) {
                    acc = sumUp(acc, sin[(l_id * 2) + i]);
                }
                sout[l_id] = acc;
            }
            barrier(CLK_LOCAL_MEM_FENCE);
            size = ( size / 2 );
            sin = ( sout==sdata3 ) ? sdata3 : sdata2;
            sout = ( sout==sdata3 ) ? sdata2 : sdata3;
        }
        int l_id = get_local_id(0);
        if (l_id < 1)
            g_odata[wg_id + l_id] = sdata2[l_id];
    }
}

```

Listing 1: OCL1: Generated OpenCL code for the low-level program LLP1

Listing 1 shows the OpenCL program OCL1 generated by our code generator [16] from the low-level program LLP1. The generator does not apply any optimizations, but rather transforms low-level patterns into imperative OpenCL code as indicated in Fig. 4 for the low-level *map* patterns. Multidimensional arrays have a flat representation in our imperative OpenCL code: therefore, no code is emitted when visiting patterns that change the data layout, like *split*, *join*, or *reorderStride*; these patterns rather influence the generation of how data is accessed by subsequent patterns.

## 5 Evaluation

In this section, we experimentally evaluate the OpenCL kernels generated from the three low-level programs LLP1-LLP3 (the latter two shown in Appendix C) which have been systematically derived from our initial high-level program *reduce* (+). Interestingly, the low-level program LLP1 describes the computation as implemented in the first version by Nvidia [8], the code for LLP2 is very similar to the fourth implementation, and the code for LLP3 corresponds to the fully optimized, seventh Nvidia’s version in the same paper.

We use an Nvidia GeForce GTX 480 GPU to conduct our experiments using the OpenCL runtime from Nvidia’s CUDA-SDK 5.5 and driver version 310.44. To measure kernel run times, we use the OpenCL profiling API and we exclude

data transfer times to focus on the quality of the generated OpenCL kernels. Each experiment is repeated 100 times, we report the median run time.

Figure 7 shows the performance of our OpenCL kernels (OCL1, OCL2, OCL3) obtained by means of formal transformations and automatic code generation as compared to the hand-written and manually tuned kernels provided by Nvidia in [8] (reduce1, reduce4, reduce7). We also compare our programs to the kernels from two libraries which implement manually optimized versions of the parallel reduction for GPUs: cuBLAS [12] and Thrust [2].

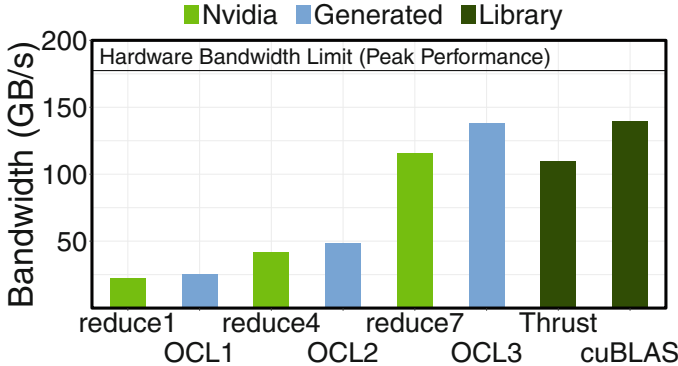


Fig. 7. Performance comparison for generated code against hand-tuned OpenCL code

In order to compare our performance to the peak performance of the GPU, we report our results as achieved bandwidth in GB/s by dividing the input data size in gigabytes by the elapsed run time in seconds. We observe in Fig. 7 that in all cases the performance of our code is on par with the performance of the corresponding manually-tuned codes from [8]. Our most optimized version OCL3 generated from LLP3 slightly outperforms the reduce7 code from [8] and the Thrust code, and we almost exactly match the performance of the cuBLAS library, which is the currently best known parallel implementation of reduction.

## 6 Conclusion

In this paper we present a transformation-based approach to developing high-performance GPU programs using patterns and rewrite rules. We introduce novel, OpenCL-specific low-level patterns to map our computations to the thread and memory hierarchy of the GPU hardware, explicitly describing implementation choices. We formalized well-known optimizations in order to systematically transform high-level programs to provably-correct, optimized low-level programs, rather than apply *ad-hoc* optimizations following the informal optimization guides from GPU vendors. We choose the OpenCL model because it allows to target a wide range of parallel accelerators. However, our transformational

programming approach is not limited to OpenCL: we may also use other models like CUDA or OpenMP.

While this paper focuses on formalizing low-level OpenCL-related patterns and rewrite rules, the order in which to apply these rules remains an open research question. Since multiple rewrite rules might be applicable at the same time and some rules can be applied infinitely often, the space of possible low-level expressions needs to be efficiently searched. An automatic randomized search strategy in [13] already leads to well performing programs. One possibility to prune the search space is to package often occurring combinations of rules in so-called macro-rules to encode specific optimizations like tiling. Analytical cost models or heuristics based on machine learning can guide the optimization process.

Experiments show that our transformation-based approach achieves performance which is competitive or even better than hand-tuned code written by performance experts and used in the modern vendor libraries for accelerators.

**Acknowledgments.** This work was supported by the German Research Council (DFG) within the Cluster of Excellence CiM (University of Münster), by the German Ministry of Education and Research (BMBF) within the project HPC<sup>2</sup>SE, and by a EuroLab-4-HPC collaboration. We thank Nvidia for their generous hardware donation used in our experiments.

## Appendix

### A Additional Rewrite Rules

$$f = \text{map } id \circ f = f \circ \text{map } id \quad (21)$$

$$f = \lambda x. f \ x \quad (22)$$

$$id = \text{join} \circ \text{split } n \quad (23)$$

$$\text{reduce } (\oplus) \circ \text{join} = \text{reduce } (\oplus) \circ \text{join} \circ \text{map } (\text{reduce } (\oplus)) \quad (24)$$

$$\text{reduce } (\oplus) = \text{reduce } (\oplus) \circ \text{join} \circ \text{map } (\text{reduce } (\oplus)) \circ \text{split } m \quad (25)$$

### B Proof of a Rewrite Rule

Rewrite rules are proved using equational reasoning. As an example we prove rule (25) which introduces layers in the computation hierarchy of a reduction: first a partial reduction is computed, followed by a reduction combining all temporary results.

*Proof (Reduce-Promotion Variant).* Let  $n$  be a number divisible by  $m$ .

$$\begin{aligned}
& (\text{reduce } (\oplus) \circ \text{join} \circ \text{map } (\text{reduce } (\oplus)) \circ \text{split } m) [x_1, \dots, x_n] \\
\{ \text{Def. split (4)} \} & \\
& = (\text{reduce } (\oplus) \circ \text{join} \circ \text{map } (\text{reduce } (\oplus))) [[x_1, \dots, x_m], \dots, [x_{n-m}, \dots, x_n]] \\
\{ \text{Def. map (1)} \} & \\
& = (\text{reduce } (\oplus) \circ \text{join}) [\text{reduce } (\oplus) [x_1, \dots, x_m], \dots, \text{reduce } (\oplus) [x_{n-m}, \dots, x_n]] \\
\{ \text{Def. reduce (2)} \} & \\
& = (\text{reduce } (\oplus) \circ \text{join}) [[x_1 \oplus \dots \oplus x_m], \dots, [x_{n-m} \oplus \dots \oplus x_n]] \\
\{ \text{Def. join (5)} \} & \\
& = \text{reduce } (\oplus) [x_1 \oplus \dots \oplus x_m, \dots, x_{n-m} \oplus \dots \oplus x_n] \\
\{ \text{Def. reduce (2), associativity of } \oplus \} & \\
& = [x_1 \oplus \dots \oplus x_m \oplus \dots \oplus x_{n-m} \oplus \dots \oplus x_n] \\
\{ \text{Def. reduce (2)} \} & \\
& = \text{reduce } (\oplus) [x_1, \dots, x_n] \quad \square
\end{aligned}$$

## C Derived Low-Level Reduction Programs

$$\begin{array}{l}
\text{reduce } (+) \circ \text{join} \circ \text{mapWorkgroup } ( \\
\quad \text{join} \circ \text{toGlobal } (\text{mapLocal } (\text{mapSeq } \text{id})) \circ \text{split } 1 \circ \\
\quad (\lambda \text{ xs} . \text{join} \circ \text{mapLocal } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } ((\text{size } \text{x s})/2) \text{ xs})^7 \circ \quad (\text{LLP2}) \\
\quad \text{join} \circ \text{toLocal } (\text{mapLocal } (\text{reduce } (+))) \circ \text{split } 2 \circ \\
\quad \text{reorderStride } 128) \circ \text{split } (2 \times 128) \\
\hline
\text{reduce } (+) \circ \text{join} \circ \text{mapWorkgroup } ( \text{join} \circ \text{toGlobal } (\text{mapLocal } (\text{mapSeq } \text{id})) \circ \text{split } 1 \circ \text{join} \circ \text{mapWarp } ( \\
\quad \text{join} \circ \text{mapLane } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } 1 \circ \\
\quad \text{join} \circ \text{mapLane } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } 2 \circ \\
\quad \text{join} \circ \text{mapLane } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } 4 \circ \\
\quad \text{join} \circ \text{mapLane } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } 8 \circ \quad (\text{LLP3}) \\
\quad \text{join} \circ \text{mapLane } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } 16 \circ \\
\quad \text{join} \circ \text{mapLane } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } 32 \\
) \circ \text{split } 64 \circ \text{join} \circ \text{mapLocal } (\text{reduce } (+)) \circ \text{split } 2 \circ \text{reorderStride } 64 \text{ join} \circ \\
\quad \text{toLocal } (\text{mapLocal } (\text{reduce } (+))) \circ \text{split } (\text{blockSize}/128) \circ \text{reorderStride } 128) \circ \text{split } \text{blockSize}
\end{array}$$

**Fig. 8.** Two more low-level programs implementing parallel reduction. They are equivalent to the fourth and the (seventh) most optimized version described in [8], correspondingly

## References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: Fastflow: high-level and efficient streaming on multi-core. In: Programming Multi-core and Many-core Computing Systems. Wiley-Blackwell, Hoboken (2011)
2. AMD: Bolt C++ Template Library
3. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Commun. ACM **21**(8), 613–641 (1978)
4. Bird, R.S.: Algebraic identities for program calculation. Comput. J. **32**(2), 122–126 (1989)

5. Burstall, R.M., Darlington, J.: A transformation system for developing recursive programs. *J. ACM* **24**(1), 44–67 (1977)
6. Chakravarty, M., Keller, G., Lee, S., McDonell, T.L., Grover, V.: Accelerating Haskell array codes with multicore GPUs. In: DAMP, pp. 3–14. ACM (2011)
7. Gorlatch, S., Cole, M.: Parallel skeletons. In: Padua, D. (ed.) *Encyclopedia of Parallel Computing*, pp. 1417–1422. Springer, Boston (2011). <https://doi.org/10.1007/978-0-387-09766-4>
8. Harris, M., et al.: Optimizing parallel reduction in CUDA. *NVIDIA Developer Technol.* **2**(4), 1–39 (2007)
9. Holk, E., Byrd, W.E., Mahajan, N., Willcock, J., Chauhan, A., Lumsdaine, A.: Declarative parallel programming for GPUs. In: PARCO, pp. 297–304 (2011)
10. Khronos OpenCL Working Group: The OpenCL Specification
11. Kuchen, H.: A skeleton library. In: Monien, B., Feldmann, R. (eds.) *Euro-Par 2002*. LNCS, vol. 2400, pp. 620–629. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45706-2\\_86](https://doi.org/10.1007/3-540-45706-2_86)
12. Nvidia: CUDA Basic Linear Algebra Subroutines (cuBLAS). Version 6.5
13. Steuwer, M., Fensch, C., Lindley, S., Dubach, C.: Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance openCL code. In: ICFP, pp. 205–217. ACM (2015)
14. Steuwer, M., Gorlatch, S.: High-level programming for medical imaging on multi-GPU systems using the skelCL library. In: *Procedia Computer Science, ICCS*, vol. 18, pp. 749–758. Elsevier (2013)
15. Steuwer, M., Kegel, P., Gorlatch, S.: SkelCL: a portable skeleton library for high-level GPU programming. In: HIPS @ IPDPS, pp. 1176–1182. IEEE (2011)
16. Steuwer, M., Rummel, T., Dubach, C.: Lift: a functional data-parallel IR for high-performance GPU code generation. In: CGO, pp. 74–85. ACM (2017)
17. Svensson, J., Sheeran, M., Claessen, K.: Obsidian: a domain specific embedded language for parallel programming of graphics processors. In: Scholz, S.-B., Chitil, O. (eds.) *IFL 2008*. LNCS, vol. 5836, pp. 156–173. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-24452-0\\_9](https://doi.org/10.1007/978-3-642-24452-0_9)

# Domain Engineering the Magnolia Way

Magne Haveraaen<sup>(✉)</sup>

Bergen Language Design Laboratory, Department of Computer Science,  
University of Bergen, Bergen, Norway

Magne.Haveraaen@ii.uib.no

<https://bldl.ii.uib.no/>

**Abstract.** Domain engineering can be seen as the process of identifying a domain API, providing its semantics, and structuring it. The domain API is a natural framework for software product line requirements and development. Magnolia is an integrated programming and algebraic specification language. As such it gives a strong focus on API design. The Magnolia way to domain engineering can be scaled from a lightweight to a formalistic heavyweight approach. Defined APIs can easily be extended as more of the domain is investigated. This paper summarises the Magnolia domain engineering process.

## 1 Introduction

When developing a suite of software products, Software product line (SPL) engineering has over the past few decades been affirmed as an order of magnitude more efficient than conventional single product engineering [16, 17]. If done well, SPL delivers reduced cost, shorter time to market, and better quality. Central to SPL are the core assets, which includes the software resources being reused across the product line.

*Domain engineering* (DE) is the process of discovering such core assets. DE is especially effective when approaching a domain for a fresh perspective, rather than trying to reengineer existing software. *The purpose of domain engineering is to identify, model, construct, catalog, and disseminate artefacts that represent the commonalities and differences within a domain*<sup>1</sup>.

Dines Bjørner has been prominent in promoting domain engineering as a discipline since the 1970s [5], and has formulated the slogan [6]: *before hardware and software systems can be designed and coded we must have a reasonable grasp of “its” requirements; before requirements can be prescribed we must have a reasonable grasp of “the underlying” domain*. In a different context Bjørner has claimed that grasping a domain is characterised by the ability to formalise it.

In a software context, domain engineering is defining artefacts that can be represented on a computer. Computer artefacts are ultimately data structures and algorithms, which are abstracted as types and operations, respectively. A

---

<sup>1</sup> Taken from <http://www.domainengineering.org> spring 2012. This URL is now under control of an internet domain warehouse and has totally irrelevant content.

collection of types and operations with semantic exegesis form an application programmer interface (API), called a *concept* in Magnolia. The API of a domain represents a *domain specific language* (DSL), the notation we need for expressing problems and solutions within the domain. A well designed API may speed up software productivity by more than a magnitude and vastly increase the flexibility of the software products: essentially giving the developer an edge in the software and consultancy market for the domain.

The paper's contribution is a lightweight approach to domain engineering, the *Magnolia way*.

This presentation is based on Magnolia, an integrated algebraic specification and programming language being developed at Bergen Language Design Laboratory (BLDL). The Magnolia design follows the principles of Goguen-Burstal institutions [10]. Any domain analysis the Magnolia way will thus satisfy Bjørner's criterion of being a *formal concept analysis* [6], just as we achieved when analysing the numerical domain of partial differential equations [11]. We will not provide an introduction to Magnolia here. The paper contains excerpts of Magnolia specifications and code, explained in the surrounding text. The Magnolia notation follows the widely used expression and procedural statement sequence pattern, reminiscent of Pascal and C/C++/Java. The domain engineering approach itself is not tied to any particular language. Most programming languages with features for API declaration and property assertion are suitable.

This paper is organised as follows. Next we discuss the notion of domain expertise. Section 3 introduces our running example. Then we present the Magnolia domain engineering steps. Finally we conclude.

## 2 Domain Expertise

Domain engineering starts with examining the body of knowledge of the domain in question. In a scientifically well researched area, there will be a significant amount of literature (in print and available online) explaining aspects of the domain. This is in many ways a definitive body of knowledge.

Given a less developed, more user oriented domain, the classical tools from requirements engineering are good for analysing a domain. This includes developing ontologies, scenario descriptions, user stories, use cases, work flow descriptions, stakeholder surveys, etc.

Another example is when the domain analyst is also domain expert and stakeholder. Many startups are created like this. The entrepreneurs are familiar with a domain and the business idea is to provide software based on this insight. Internal domain expertise may result in an incomplete analysis as perceived shortcuts are taken rather than doing a deeper investigation of the domain.

The input from the domain expertise should not be taken literally when investigating the domain [12, 14]. The difficulty is taking what is asked as a reflection of what is actually needed, and then formulating the latter.



### 3 Running Example: 101 Companies

To show some of the technologies for domain engineering we will use the 101 Companies project [7]. Problem descriptions and other resources are available online<sup>2</sup>. Our example domain engineering is based on the reader’s general problem domain insight from the following excerpt.

The 101 system (or just “the system”) is an imaginary Human resource management system (HRMS) which serves as the “running example” in the 101 project. That is, “contributions” to the project are meant to implement or model or otherwise exercise the system for a conceived company as a client.

The system is supposed to model the company structure in terms of employees and possibly the hierarchical structure of departments. Employees are modelled in terms of their names, addresses, salaries, and possibly additional properties. The system is supposed to meet certain functional requirements such as totalling all salaries in the company.

From the above description, it is possible to identify that the HRMS systems has notions like company, person, employee, name (both for person and for company), address (both for person and for company), and salary. Each of these can be considered *data types* being used in the system. It seems natural to include national identity number<sup>3</sup> (NIN) for identifying people (like *fødselsnummer* in Norway). The notion *unique identifiers*, like NIN, is somewhat tricky. They can never be universally unique, but are always unique relative to some register.

Further elaborating the HRMS domain: Since a company has employees, it must be able to hire (and fire) people. We probably want to be able to ask if a specific person is employed, e.g., using the NIN, and if so, retrieve the employee’s person data and salary.

### 4 Steps in a Domain Engineering Process

The steps presented here represent an approach for domain engineering. They are a guide, not a straight jacket, for the domain engineering process. Each step is associated with a *probe* into the domain.

1. Determine the signature of the domain.  
What are the types and operations of the domain?
2. Expand the signature to a formal API specification.  
What are the properties of the types and operations, and how should we modularise them?
3. Refine the domain concepts using specification theory insight.  
What are the implications of the domain’s properties—do they relate to well known specifications?

<sup>2</sup> <http://101companies.org/wiki/@system>.

<sup>3</sup> [https://en.wikipedia.org/wiki/National\\_identification\\_number](https://en.wikipedia.org/wiki/National_identification_number).

4. Develop a generic support library and its architecture for the domain API.  
How to organise the domain software for generic reuse?
5. Develop applications for the domain.  
How to configure the domain library to build useful applications?

In every step the rigour and completeness of the domain engineering work can be adapted to the resources available (e.g., person months, competence, deadline).

A sketchy signature with a light level of formalisation and cursory architecture analysis, brings the process fast forward towards application development. If the appropriate parts of the domain API are provided as library components, then the process may move ahead to a deliverable application without much difficulty. Additional applications can be developed by iteratively improving the size and maturity of the API and library artefacts. This is the ideal combination of *lean business* with *software product lines* [16]. However, if the chosen API is not the epitome, the iteration for a next application may yield a materially different architecture, significantly reducing the value of the accelerated process.

A more thorough process ensures a deeper understanding of the domain, its API and architecture, and thus heightens the reusability of the defined artefacts. The more resources that can be spent in the earlier steps, the better payback in the later steps. Getting the implementation architecture right is the clue. Implementing the architecture can be done piecemeal, according to the needs of the actual applications.

The architecture design should be based on a broad understanding of the domain. This can be achieved rather informally after identifying the domain's signatures. Documentation comments for the types and operations go a long way in communicating their intention. Gradually providing more formal descriptions for the API ensures that the domain is well comprehended.

The formal API specifications from the earlier steps take on the rôle of high quality parameterised test oracles in the implementation steps [1]. Such test oracles can be used to manage the library and application development along the lines of *test driven development* (TDD) [2].

#### 4.1 Step 1: Finding the Signature

A signature is a collection of type and operation declarations, the abstractions of data structures and algorithms. Identifying the signature for a domain starts by asking the simple questions:

- What are the types in the domain?
- What are the operations?
- What information is needed and produced by each operation?

These questions build a signature—a vocabulary—for the domain concepts. Once a vocabulary is starting to emerge, it should be tried: use it to describe important problems and sketch solutions in the domain (steps 4&5). This check ensures the signature is useful for dealing with the domain's problem space. Often such

checks reveal the need to enlarge the vocabulary in order to easily express the problems and their solutions.

Choosing good names for the types and operations helps with the intuition of the vocabulary. It is important to nail the intuition down by expeditiously providing natural language definition of these entities. Such a description may take the form of a documentation comment attached to each declaration. Given our rudimentary analysis of 101 companies in Sect. 3, we can write the following Pascal style declarations of the `Person` type and access operations. The Java notation for documentation comments `/** .. */` is being used. The *getters* access properties that we assume a person must possess.

```

1  /** Information about a person: name, address, unique NIN, etc. */
2  type Person;
3  /** Name of a person, no further structure indicated. */
4  type PersonName;
5  /** Address for a person, no further structure indicated. */
6  type Address;
7  /** National identity number, a unique identification number
8   * according to a national registry of residents and citizens. */
9  type NIN;
10
11 /** Getters for the components of a Person. */
12 function getName ( p:Person ) : PersonName ;
13 function getAddress ( p:Person ) : Address ;
14 function getNIN ( p:Person ) : NIN ;
15 /** Setters for the components of a Person, but not for the NIN. */
16 function setName ( p:Person, n:PersonName ) : Person ;
17 function setAddress ( p:Person, a:Address ) : Person ;

```

Here we do not provide a setter for the NIN. We assume the NIN is immutable, even if the person moves or decides to change his/her name. This signature is open-ended. The lack of a getter for gender or birthdate does not prevent us from adding these later. We can consciously decide to not include these in our first signature analysis due to limited resources, e.g., inadequate time for in depth analysis (or page constraint for a scientific paper).

Existing concepts, such as numbers or strings, should be used wherever relevant for the domain. However, it is normally better to leave a type abstract unless there are compelling reasons otherwise. For instance, names and addresses may be represented as strings, and (in Norway at least) NINs by a sufficiently large integer type. Leaving these types abstract allow us to investigate them in more depth at a later stage. We may later find that, e.g., addresses have a country dependent structure, such as street name and house number, postal code etc. By not fixing the information content of these types now, we create a software architecture that allows us to adapt the forthcoming applications worldwide.

We should also provide operations on the company type (below), the salary type (Sect. 4.2), and other types that appear during this analysis.

```

1  /** Information on a company: who works there, salary expenses, etc. */
2  type Company;

```

```

3  type CompanyName;
4  type Salary;
5  function getCompanyName ( c:Company ) : CompanyName ;
6  function getCompanyAddress ( c:Company ) : Address ;
7  /** Is a person hired by the company (check by NIN). */
8  predicate isHired ( c:Company, n:NIN );
9  /** Get person data, salary etc of an employee, lookup by NIN. */
10 function findEmployee ( c:Company, n:NIN ) : Person
11     guard isHired(c,n);
12 function findSalary ( c:Company, n:NIN ) : Salary
13     guard isHired(c,n);
14 /** Hire a new person. The NIN is part of the Person data. */
15 function hire ( c:Company, p:Person, s:Salary ) : Company
16     guard ! isHired( c, getNIN(p) );
17 /** Fire an employee. */
18 function fire ( c:Company, n:NIN ) : Company guard isHired(c,n);

```

The predicate `isHired` checks if the NIN belongs to a registered employee. This boolean function is used as preconditions (guards) for the other operations. We also introduced a new type for the name of a company, just in case company names have a different structure from person names. On the other hand we assume company addresses have the same format as those for people. Just declaring this signature forces us to be aware of small but important design decisions. Here we decided that a salary is part of the work agreement between employer and company, rather than a personality trait. Thus the hiring function must have person and salary as separate arguments, and we retrieve these with separate NIN-indexed find operations. Maybe we should have introduced a type `Contract`, of which `Salary` was a component, and then used `Contract` instead of `Salary` in the hire and find salary/contract operations. For now, such analysis may be too heavyweight. Besides, introducing `Contract` would not change the structure of the operations for `Company`.

A signature contains just declarations and document comments. Hence the cost of modifying it will be low. Useful and needed modifications to the signature are often discovered during the later domain engineering steps. For instance, when formalising the API (steps 2&3), or checking the API's applicability for expressing domain problems and applications (steps 4&5). Then the signature analysis should be reiterated in order to declare more types and operations. A difficult issue to discover is when two or more distinct operations subconsciously get the same name. For instance, the word *hire* can coincidentally be used for both *rent* and *employ*, possibly causing confusion if the company was in the business of subletting apartments. The separate rôles often blocks our intuition from discovering the naming collision. Writing the documentation comment may give a hint: it becomes difficult to formulate since it has to capture confusing behaviour. Yet, even if the issue is overlooked during the signature analysis, the next step of formalising the API will often expose such conflicts.

## 4.2 Step 2: Formalising the Specification

In the previous step we identified the types and operations and the operations' information flow. This gives a surprising amount of insight and spawns many high level design decisions. In the second step we try to formalise our understanding of the domain's API.

- How do we organise the types and operations as a collection of cohesive *concepts*?
- How do the operations relate to each other, i.e., what are the *axioms*?

Such APIs are called *concepts*, a name introduced by Stepanov [15].

Grouping of the declarations as concepts is important to keep the size of each concept manageable. Simple concepts hold one or a few types and their most important operations. Operations that relate several types are often placed in concepts separate from the operations on a single type. Different aspects of a type can be explored in separate concepts. The same goes for axioms.

In Magnolia an axiom has a predicate style declaration, but the body is a sequence of procedural style statements with (a high proportion of) *assert* statements. The assertions are to hold for every combination of data values for the axiom's argument types. Values violating guards for the axiom or the operations in the assert expressions are ignored. Normally an assert expression is a boolean expression as in a programming language. This is deliberate. It provides software developers with a familiar notation with familiar semantics for the assertions. The axiom language can extend beyond this, e.g., encompassing full first order logic.

In our 101 companies example, we will group the operations on `Person` in one concept, those on `Company` in another, and so forth.

```

1  concept SimplePerson = {
2    /** Information on a person: name, address, unique (per country) NIN. */
3    type Person;
4    type PersonName;
5    type Address;
6    type NIN;
7    /** Getters for the components of a Person. */
8    function getName ( p:Person ) : PersonName ;
9    function getAddress ( p:Person ) : Address ;
10   function getNIN ( p:Person ) : NIN ;
11   /** Setters for the components of a Person. */
12   function setName ( p:Person, n:PersonName ) : Person ;
13   function setAddress ( p:Person, a:Address ) : Person ;
14
15   axiom getSetNameAxiom ( p:Person, n:PersonName ) {
16     assert getName(setName(p,n)) == n;
17     assert getAddress(setName(p,n)) == getAddress(p);
18     assert getNIN(setName(p,n)) == getNIN(p);
19   };
20   axiom getAddressAxiom ( p:Person, a:Address ) {
```

```

21     assert getName(setAddress(p,a)) == getName(p);
22     assert getAddress(setAddress(p,a)) == a;
23     assert getNIN(setAddress(p,a)) == getNIN(p);
24   };
25 };

```

The axioms here state that a setter only modifies one component of the person's data with respect to the declared getters. Elaborations of the person name, address and NIN types belong in separate concepts, one for each of these types. The simple company concept below does not include all the company operations declared above. The remaining operations and axioms are placed in other concepts.

```

1  concept SimpleCompany = {
2    type Person; type NIN; function getNIN ( p:Person ) : NIN ;
3
4    type Company;
5    type CompanyName;
6    type Address;
7    type Salary;
8    function getCompanyName ( c:Company ) : CompanyName ;
9    function getAddress ( c:Company ) : Address ;
10   predicate isHired ( c:Company, n:NIN );
11   /** Find person data, salary etc of an employee, lookup by NIN. */
12   function findEmployee ( c:Company, n:NIN ) : Person
13     guard isHired(c,n);
14   function findSalary ( c:Company, n:NIN ) : Salary
15     guard isHired(c,n);
16   function hire ( c:Company, p:Person, s:Salary ) : Company
17     guard ! isHired ( c, getNIN(p) );
18   axiom getHireAxiom ( c:Company, p:Person, s:Salary )
19     guard ! isHired(c,getNIN(p)) {
20     var cprime = hire(c,p,s);
21     assert getCompanyName(cprime) == getCompanyName(c);
22     assert getAddress(cprime) == getAddress(c);
23     assert isHired(cprime,getNIN(p));
24     assert findEmployee(cprime,getNIN(p)) == p;
25     assert findSalary(cprime,getNIN(p)) == s;
26   };
27 };

```

Here `getHireAxiom` groups many individual assertions. This can be convenient if they have the same arguments and are otherwise closely related.

We can combine concepts and declarations when defining a new concept. Here we are also using a renaming mechanism to replace the name (in the context of this concept) of the salary type to `Contract`. The renaming is a list of name substitutions in square brackets.

```

1  concept SimpleCP = {
2    use SimplePerson;
3    use SimpleCompany[ Salary => Contract ];
4    function fire ( c:Company, n:NIN ) : Company guard isHired(c,n);

```

```

5  /** What happens when someone is fired after a hiring has taken place. */
6  axiom fireHireAxiom ( c:Company, p:Person, k:Contract, nin:NIN )
7  guard ! isHired( c, getNIN(p) ) {
8  if getNIN(p) == nin then
9    /* Firing the recent hiree has no net effect on the company. */
10   assert fire( hire(c,p,k), nin ) == c;
11 else
12   /* Interchangeability of firing someone and hiring somebody else. */
13   assert fire( hire(c,p,k), nin ) == hire( fire(c,nin), p, k );
14 end;
15 };
16 };

```

The **use** statement does a straight forward inclusion of content of the named concept. The union of the included declarations, and any local declarations, is then formed. Thus two instances of the same declaration, e.g., the **Person** type declaration, is merged into one instance, and the union of all person related operations and axioms, from any declaration in scope, become operations and axioms on the merged type. The new **fireHireAxiom** relates firing an employee to the hiring of a person.

When declaring concepts, we may find that formulating the specifications become unduly complicated.

- If we find ourselves copying the same axioms many times with only modifying type and operation names, we probably have embodied the same ideas in several types and operations. This can be handled by extracting a more abstract concept, and including it, with appropriate renamings.
- If we become bewildered about how to formalise properties of an operation, or we find we are starting to write a list of case distinctions based on an operation’s arguments, this may indicate that one operation has taken several conflicting rôles. Declaring a new operation for each rôle may clarify and simplify the axioms.
- If we find that we have to use very expressive logics, like full first order, rather than, e.g., plain equations, this may indicate that it will be a good idea to introduce additional operations (and possibly additional types) enabling the use of a less expressive logic. There are hard theoretical results about adding helper (often called hidden) types and operations to a specification, in order to get away with a less expressiveness logic [3]. Less expressive logics typically have better tool support and are generally easier to reason about.
- Each assertion should be short, and should not need an extensive setup in the axiom. If the assertion becomes large, this indicates that the API should be refactored to include more, and more appropriate, operations and types.

The aim of this step is to get a concise, but useful, collection of concepts that reels in the essentials of a domain. This will often capture insight that the domain description did not make clear, or that the domain expertise never formulate consciously. The dialogue when organising concept and writing axioms may thus inspire the domain expert with new insights and reflections. This may yield an

even more cutting edge description of the domain—and ideas for unanticipated applications.

Thinking about the salary notion from this perspective, we may decide that it should be possible to add salaries (for summing monthly payments to an annual salary), and that it should be possible to adjust the salary by a factor (for relative salary increases). These operations are not part of the description in Sect. 3, but seem intrinsic to the domain.

```

1  concept Salary = {
2    type Salary;
3    type Factor;
4    function plus ( s1:Salary, s2:Salary ) : Salary;
5    function increase ( f:Factor, s:Salary ) : Salary;
6    axiom increaseAddAxiom ( f:Factor, s1:Salary, s2:Salary ) {
7      assert increase( f, plus(s1,s2) )
8        == plus( increase(f,s1), increase(f,s2) );
9    };
10 };

```

### 4.3 Step 3: Refining the Domain Concepts

After getting a grip on the APIs involved in a domain, we can start investigating the formalised concepts and their logical implications. This will benefit from insight into generic concepts, such as common mathematical abstractions. Looking at the salary concept, we may consider addition to be an abelian group (addition is associative and commutative, with zero and subtraction), factor to be an abelian monoid (associative and commutative multiplication of factors with a neutral constant one), and factor to be a monoid action on the salary type (increase salary by a factor).

```

1  concept FactorActionSalary = {
2    use Salary;
3    use Group[ T => Salary ];
4    use AbelianMonoid[ T => Factor ];
5    use MonoidAction[ T => Factor, action => increase ];
6  };

```

Reusing well investigated properties allow us to think about whether the implications that follow also hold in the domain. This strengthens our confidence that the domain is well understood. Further we may use this to simplify our domain description, e.g., by noting that some of our domain axioms or concepts are redundant, or can be derived from simpler concepts that were overlooked in the analysis so far. A well founded simplification of the domain description may give significant savings in the implementation steps.

### 4.4 Step 4: Library Development and Architecture Design

The costs of an implementation can be significantly reduced by establishing the right software architecture, an architecture that encourages reusing implementations and flexibly combining them into varied applications. Generic programming



is key to modern reuse practices. Generic code has a **requires** part (an input API), and provides an (output) API build on the requirements.

Consider an implementation for the Company concept. It must obviously contain the personnel and the salaries of each hiree. Each must be easily accessible by the NIN. A dictionary type provides lookup of data by a key (the NIN). The company data structure must also have fields for company name and address. We probably want to provide the company with a managing director (CEO in US terminology), though this is not explicitly required by the specification so far.

```

1  implementation simpleCompany = {
2    /** Require person operations and rudimentary salary and NIN types. */
3    require SimplePerson;
4    /** Require atomic type for company name. */
5    require type CompanyName;
6    /** Require useful operations on a NIN-indexed dictionary of personnel. */
7    require Dictionary
8      [ Key => NIN, Data => Person, Dictionary => Personnel ]
9      [ createDictionary => createPersonnel ];
10   /** Require useful operations on a NIN-indexed dictionary of contracts. */
11   require Dictionary
12     [ Key => NIN, Data => Contract, Dictionary => Contracts ]
13     [ createDictionary => createContracts ];
14
15   /** Define the company attributes: induces getters, setters, ... */
16   type Company =
17     struct{
18       var companyName:CompanyName; var companyAddress:Address;
19       var md:NIN;
20       var personnel:Personnel; var contracts:Contracts; };
21   /** Creating the company using the declaration's attribute constructor. */
22   function createCompany
23     ( cn:CompanyName, ca:Address, md:Person, k:Contract ) : Company
24   = hire(
25     makeCompany
26       (cn,ca,getNIN(md),createPersonnel(),createContracts()),
27     md, k);
28   predicate isHired ( c:Company, n:NIN ) =
29     isPresent( getPersonnel(c), n );
30   function findEmployee ( c:Company, n:NIN ) : Person
31     guard isHired(c,n) = find( getPersonnel(c), n );
32   function findContract ( c:Company, n:NIN ) : Contract
33     guard isHired(c,n) = find( getContracts(c), n );
34   function hire ( c:Company, p:Person, k:Contract ) : Company
35   guard ! isHired ( c, getNIN(p) ) = {
36     var ps = insertNew(getPersonnel(c),getNIN(p),p);
37     var ks = insertNew(getContracts(c),getNIN(p),k);
38     return setPersonnel(setContracts(c,ks),ps);
39   };
40   function fire ( c:Company, n:NIN ) : Company

```

```

41  guard isHired(c,n) = {
42    var cp = setPersonnel( c, removeEntry(getPersonnel(c),n) );
43    cp = setContracts ( cp, removeEntry(getContracts(cp),n) );
44    return cp;
45  };
46 };

```

The *require* clause for a concept (or type or operation) makes the types and operations (but not the axioms) available as a required API for the implementation. These are the generic parameters of the implementation. In order to claim that the implementation above satisfies the `SimpleCF` concept, we also need to make certain any generic arguments have the right properties.

```

1  satisfaction simpleCompany_models_SimpleCP = {
2    use SimplePerson;
3    type CompanyName;
4    use Dictionary
5      [ Key => NIN, Data => Person, Dictionary => Personnel ]
6      [ createDictionary => createPersonnel ];
7    use Dictionary
8      [ Key => NIN, Data => Contract, Dictionary => Contracts ]
9      [ createDictionary => createContracts ];
10 } with simpleCompany_models SimpleCP;

```

This separates the correctness argument from the code itself, following the institution framework [10] closely. It breaks with the dominant tradition from [9,13] where the specification of software components is integrated with the implementation.

The specification of the premises for the generic implementation, should now make it possible to prove the correctness of the implementation with respect to the target specification. We can also use axioms as test oracles, by providing the premises as concrete implementations. In the satisfaction relation above, the `SimplePerson` part of `SimpleCP` is satisfied by the premises. In the simplified setting below, we use the axioms from `SimpleCompany`, avoiding the extra layer of `SimplePerson` axioms.

```

1  renaming SC = [ CompanyName => Integer,
2    NIN => Integer, Person => Integer, Address => Integer ];
3
4  satisfaction test_simpleCompany_models_SimpleCompany_Integer = {
5    use modulusInteger8bitCxx;
6    use dictionaryArrayLinearCxx[ Key => Integer, Data => Integer ];
7    /** An arbitrary link between a person and the person's NIN. */
8    function getNIN ( p:Integer ) : Integer = p - 1 ;
9    function setNIN ( p:Integer, nin:Integer ) : Integer = nin + 1 ;
10 } with simpleCompany
11   [ SC, Personnel => Dictionary, Contracts => Dictionary ]
12   [ createPersonnel => createDictionary,
13     createContracts => createDictionary ]
14   [ Contract => Integer, PersonName => Integer, getName => getNIN,

```

```

15     getAddress => getNIN, setName => setNIN, setAddress => setNIN
16   ]
17   models
18     SimpleCompany
19     [ SC, findSalary => findContract, Salary => Integer ];

```

Here we use an 8bit integer as the data type for all simple generic arguments (see the renaming `SC`), and a single (array list) dictionary data type for all the internal data bases of the company. Given the generic typing discipline of `simpleCompany_models_SimpleCP` (ignoring the axioms), and assuming a validated dictionary implementation, only testing a small part of 8bit data space is needed to guarantee correctness of the `simpleCompany` implementation [4]. The details are beyond the scope of this paper.

We can coin *proof driven design* (PDD) for deriving the implementation from expressed proof requirements, i.e., starting with a satisfaction claim as `simpleCompany_models_SimpleCP`. Starting with a concrete instantiation, i.e., from `test_simpleCompany_models_SimpleCompany_Integer`, we get a kind of test driven design (TDD) [2] which we could call *axiom driven design* (ADD).

Carefully selecting how to organise the implementation can cut down on development time. For instance, it may be worthwhile to start with a simplistic implementation (straight forward algorithms and data structures), then provide more advanced and efficient alternatives later on. The first application is a final proof-of-concept for the API. It shows that the chosen architecture and API provides a solution to the domain.

## 4.5 Step 5: Application Development

Application development is configuring the library to achieve an executable. We can produce an interactive application by adjoining a *graphical user interface* (GUI) to the application. GUIs can be created from multi-way dataflow constraint systems [8], which provides a low cost approach to advanced user interface logic and reactive behaviour. The ability to flexibly build different configurations yields a software product line of applications.

For our running example, we can instantiate the generic person and company implementations with suitable executable implementations for names (for example a UTF-8 string), for addresses (for example a triple of UTF-8 strings: street address and number, post code and city name, country name), for NINs (for example 64bit integers), and for salaries (for example a fixed digit representation with 6 decimal digits). The dictionary part (personnel and contract) can be covered by an external data base. This will be a simple HRMS engine suited for the Norwegian market. With different selections for name, address, salary and NIN types, we can create configurations suitable for other countries. If we cannot configure a distribution for some country, e.g., Russia, we need to go back to earlier steps. The nearest being step 4 which will deliver missing implementation components. For more fundamental issues, we need to regress to even earlier steps, but this is rare.

With this perspective, requirements engineering is a part of domain engineering. The functional parts of a requirement is given by the domain concepts. The non-functional parts are related to choosing suitable configurations, or providing another generic implementation with the required characteristics.

## 5 Conclusion

In this paper we have shown how the Magnolia way of domain engineering provides an agile approach to domain engineering and implementation. The thoroughness of the approach can be adjusted from fairly lightweight, mostly informal, through an in depth domain and architecture engineering heavy on formalisation.

The Magnolia way is less meticulous than Bjørner's TripTych approach to domain engineering [6], but shares the same goals—and achieves the same rigour when used as a heavyweight approach. The *proof driven* (PDD) and *axiom driven* (ADD) development methods indicated in the paper are both based on concepts formalised during the domain analysis steps. These methods are refinements of the well established agile *test driven development* (TDD) [2].

Taken together the Magnolia way delivers the core artefacts needed for supporting *software product lines* (SPL) and reap the benefits of lower development costs, shorter time to market, and better quality promised by SPL [17], while meeting the demands of lean business [16] in the domain engineered, and thus mastered, domain.

**Acknowledgment.** This research has in part been financed by The Research Council of Norway through the project Design of a Mouldable Programming Language (DMPL). The Magnolia IDE and compiler was implemented by Anya Bagge during her postdoc for DMPL.

## References

1. Bagge, A.H., David, V., Haveraaen, M.: Testing with axioms in C++ 2011. J. Object Technol. **10**(10), 1–32 (2011). <https://doi.org/10.5381/jot.2011.10.1.a10>
2. Beck, K.: Test-Driven Development: By Example. Addison-Wesley, Boston (2002)
3. Bergstra, J.A., Tucker, J.: Algebraic specifications of computable and semicomputable data types. Theor. Comput. Sci. **50**, 137–181 (1987). [https://doi.org/10.1016/0304-3975\(87\)90123-X](https://doi.org/10.1016/0304-3975(87)90123-X)
4. Bernardy, J.-P., Jansson, P., Claessen, K.: Testing polymorphic properties. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 125–144. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-11957-6\\_8](https://doi.org/10.1007/978-3-642-11957-6_8)
5. Bjørner, D.: The Vienna Development Method (VDM). In: Blum, E.K., Paul, M., Takasu, S. (eds.) Mathematical Studies of Information Processing. LNCS, vol. 75, pp. 326–359. Springer, Heidelberg (1979). [https://doi.org/10.1007/3-540-09541-1\\_33](https://doi.org/10.1007/3-540-09541-1_33)
6. Bjørner, D.: Manifest domains: analysis and description. Formal Aspects Comput. **29**(2), 1–51 (2016). <https://doi.org/10.1007/s00165-016-0385-z>

7. Favre, J.-M., Lämmel, R., Schmorleiz, T., Varanovich, A.: *101companies*: a community project on software technologies and software languages. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 58–74. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-30561-0\\_6](https://doi.org/10.1007/978-3-642-30561-0_6)
8. Foust, G., Järvi, J., Parent, S.: Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In: Kästner, C., Gokhale, A.S. (eds.) Proceedings of GPCE 2015, Pittsburgh, PA, USA, 26–27 October 2015, pp. 121–130. ACM (2015). [https://doi.org/10.1007/978-3-642-30561-0\\_6](https://doi.org/10.1007/978-3-642-30561-0_6)
9. Goguen, J.A.: Reusing and interconnecting software components. *Computer* **19**(2), 16–28 (1986). <https://doi.org/10.1109/MC.1986.1663146>
10. Goguen, J.A., Burstall, R.M.: Institutions: abstract model theory for specification and programming. *J. ACM* **39**(1), 95–146 (1992). <https://doi.org/10.1145/147508.147524>
11. Haveraaen, M., Friis, H.A., Johansen, T.A.: Formal software engineering for computational modelling. *Nordic J. Comput.* **6**(3), 241–270 (1999)
12. Hjørland, B.: Domain analysis in information science: Eleven approaches - traditional as well as innovative. *J. Doc.* **58**(4), 422–462 (2002). <https://doi.org/10.1108/00220410210431136>
13. Hoare, C.A.R.: Proof of correctness of data representations. In: Bauer, F.L., Dijkstra, E.W., Ershov, A., Griffiths, M., Hoare, C.A.R., Wulf, W.A., Samelson, K. (eds.) Language Hierarchies and Interfaces. LNCS, vol. 46, pp. 183–193. Springer, Heidelberg (1976). [https://doi.org/10.1007/3-540-07994-7\\_54](https://doi.org/10.1007/3-540-07994-7_54)
14. Robertson, J.: Eureka! Why analysts should invent requirements. *IEEE Softw.* **19**(4), 20–22 (2002). <https://doi.org/10.1109/MS.2002.1020281>
15. Stepanov, A., McJones, P.: Elements of Programming, 1st edn. Addison-Wesley Professional, Boston (2009)
16. Terho, H., Suonsyrjä, S., Jaaksi, A., Mikkonen, T., Kazman, R., Chen, H.: Lean startup meets software product lines: Survival of the fittest or letting products bloom? In: Nummenmaa, J., Sievi-Korte, O., Mäkinen, E. (eds.) Proceedings of SPLST 2015. CEUR Workshop Proceedings, vol. 1525, pp. 134–148. CEUR-WS.org (2015). <http://ceur-ws.org/Vol-1525/paper-10.pdf>
17. Tüzün, E., Tekinerdogan, B.: Analyzing impact of experience curve on ROI in the software product line adoption process. *Inf. Softw. Technol.* **59**, 136–148 (2015). <https://doi.org/10.1016/j.infsof.2014.09.008>

# Approximating Event System Abstractions by Covering Their States and Transitions

Jacques Julliand, Olga Kouchnarenko, Pierre-Alain Masson<sup>(✉)</sup>,  
and Guillaume Voiron

FEMTO-ST, UMR 6174 CNRS, Univ. Bourgogne Franche-Comté,  
16, route de Gray, 25030 Besançon Cedex, France  
{jjulliand,okouchna,pamasson,gvoiron}@femto-st.fr

**Abstract.** In event systems, contrarily to sequential ones, the control flow is implicit. Consequently, their abstraction may give rise to disconnected and unreachable paths. This paper presents an algorithmic method for computing a reachable and connected under-approximation of the abstraction of a system specified as an event system. We compute the under-approximation with concrete instances of the abstract transitions, that cover all the states and transitions of the predicate-based abstraction. To be of interest, these concrete transitions have to be reachable and connected to each other. We propose an algorithmic method that instantiates each of the abstract transitions, with heuristics to favour their connectivity. The idea is to prolong whenever possible the already reached sequences of concrete transitions, and to parameterize the order in which the states and actions occur. The paper also reports on an implementation, which permits to provide experimental results confirming the interest of the approach with related heuristics.

**Keywords:** Predicate abstraction · Under-approximation  
Event systems

## 1 Introduction

Abstracting a program or its specification allows to control the size of its state space description, at the price of a loss in accuracy. That facilitates their algorithmic exploitation, otherwise limited by the huge if not infinite number of concrete states. The general idea of abstraction is to gather states that share common properties into super-states. In predicate abstraction [1] the concrete states are mapped onto a finite set of abstract ones, by means of a set of predicates that characterizes each abstract state. An abstract transition links two abstract states when it has at least one concrete instantiation. Such transitions are called *may* [2], meaning that they may be instantiated. Still there is no guarantee that two consecutive *may* transitions can necessarily be instantiated as two consecutive *connected* concrete transitions: their respective target and source concrete states may differ.

This paper aims at computing connected and reachable concrete paths from a predicate abstraction of a system formally specified as an event system [3], which is a special kind of action system [4,5]. We propose an algorithmic method for computing an under-approximation that covers all the states and transitions of this abstraction.

An event in an event system specifies state variable modifications by means of a guarded action. The actions are activated whenever their guard becomes true, so that there is no natural control flow as in a program. As a result, paths of the system may become disconnected and even unreachable in the abstraction. Still, we are interested in covering the reachable part of the abstraction as best as possible. This work has been motivated by a testing purpose: our aim is to cover by tests some selected execution paths of a system, and abstracting it avoids its state space to blow-up. But the method could also apply for example to the model-checking of safety properties.

We propose to under-approximate the abstraction by computing concrete instances of the abstract event sequences. The idea behind our method is to favour the connectivity and reachability of the successive concrete instances by prolonging whenever possible the already reached concrete transitions. Our proposal in this paper is as sketched:

- we instantiate each of the abstract transitions by enumerating all the possibilities of connecting two abstract states by any event,
- we use heuristics for controlling the order in which the events and states are enumerated, according to some know-how of the natural flow of the events succession,
- we use concrete state coloration, similarly to [6], for prolonging preferably the sequences known to be reachable and connected.

Our contributions allow then generating a concrete transition system from an event system. We also report on an implementation, which permits us to provide experimental results confirming the interest of the approach with the related heuristics.

The technical background of our paper regarding event systems, predicate abstraction and may transition systems is given in Sect. 2. Section 3 presents an electrical system as an illustrative example. The algorithm for computing both an abstraction and its approximation is presented in Sect. 4. The heuristics that we propose to enhance the coverage achieved by the algorithm are given in Sect. 5. Our experimental results are presented in Sect. 6. Section 7 describes related work, and Sect. 8 concludes the paper.

## 2 Background

In this paper systems are specified by event systems (ES) described in the B syntax [3,7]<sup>1</sup>. Notice however that our proposals and results are generic since event system semantics is defined by concrete labelled transition systems.

<sup>1</sup> Our experimental models were written in B, but could alternatively be translated to a syntax with guarded commands [4], such as Abstract State Machines [8,9].

In this section we first present the syntax and the semantics of the B event systems. Then we present the concept of predicate abstraction and formalize the abstraction of event systems by means of May Transition Systems (MTS).

## 2.1 Model Syntax and Semantics

We start by introducing B event systems in Definition 1. The events are defined by means of guarded actions [4] by composition of five primitive actions where  $a$ ,  $a_i$  are actions,  $E$ ,  $F$  are arithmetic expressions and  $P$ ,  $P'$  are predicates: *skip* an action with no effect,  $x, y := E, F$  a multiple assignment,  $P \Rightarrow a$  a guarded action,  $a_1 \parallel a_2$  a bounded non-deterministic choice between  $a_1$  and  $a_2$ , and  $@z.a$  an unbounded non-deterministic choice  $a_{z_1} \parallel a_{z_2} \parallel \dots$  for all the values of  $z$  satisfying the guard of  $a$  denoted as  $\text{grd}(a)$ . Here  $\text{grd}$  is defined on the primitive actions by:  $\text{grd}(\text{skip}) \stackrel{\text{def}}{=} \text{true}$ ,  $\text{grd}(x, y := E, F) \stackrel{\text{def}}{=} \text{true}$ ,  $\text{grd}(P' \Rightarrow a) \stackrel{\text{def}}{=} P' \wedge \text{grd}(a)$ ,  $\text{grd}(a_1 \parallel a_2) \stackrel{\text{def}}{=} \text{grd}(a_1) \vee \text{grd}(a_2)$  and  $\text{grd}(@z.a) \stackrel{\text{def}}{=} \exists z. \text{grd}(a)$ .

**Definition 1 (Event System).** *Let  $\text{Ev}$  be a set of event names. A B event system is a tuple  $\langle X, I, \text{Init}, \text{EvDef} \rangle$  where  $X$  is a set of state variables,  $I$  is a state invariant,  $\text{Init}$  is an initialization action such that  $I$  holds in any initial state,  $\text{EvDef}$  is a set of event definitions, each in the shape of  $e \stackrel{\text{def}}{=} a$  for any  $e \in \text{Ev}$ , and such that every event preserves  $I$ .*

Following [10], we use labelled transition systems to define the semantics of event systems. An example of a B event system will be provided in Sect. 3.

Let  $e \stackrel{\text{def}}{=} a$  be an event. It has a *weakest precondition* [5] and a *weakest conjugate precondition* [10] w.r.t. a set of target states  $Q'$  denoted respectively as  $wp(a, Q')$  and  $wcp(a, Q')$ .  $wp(a, Q')$  is the largest set of states from which applying  $a$  always leads to a state of  $Q'$  whereas  $wcp(a, Q')$  is the largest set of states from which it is possible to reach a state of  $Q'$  by applying  $a$ . An event also defines a relation between the values of the state variables  $X$  before and after the application of the event. It is expressed by the before-after predicate of the event  $e \stackrel{\text{def}}{=} a$  denoted as  $\text{prd}_X(a)$ .

Let us now formally define  $wp$ ,  $wcp$  and  $\text{prd}_X$  following [11]. Classically, we directly consider the set of states  $Q$  and  $Q'$  as predicates of the same name: a set of states  $Q$  defines a predicate  $Q$  that holds in any state of  $Q$  but does not holds in any state not in  $Q$ .

We define the  $wp$  w.r.t. the five primitive actions as:

- $wp(\text{skip}, Q') \stackrel{\text{def}}{=} Q'$ ,
- $wp(x := E, Q') \stackrel{\text{def}}{=} Q'[E/x]$  that is the usual substitution of  $x$  by  $E$ ,
- $wp(P \Rightarrow a, Q') \stackrel{\text{def}}{=} P \Rightarrow wp(a, Q')$ ,
- $wp(a_1 \parallel a_2, Q') \stackrel{\text{def}}{=} wp(a_1, Q') \wedge wp(a_2, Q')$ ,
- $wp(@z.a, Q') \stackrel{\text{def}}{=} \forall z. wp(a, Q')$  where  $z$  is only bound by predicates in  $a$ .



We define the  $wcp$  and  $prd_X$  w.r.t.  $wp$  as:

- $wcp(a, Q') \stackrel{\text{def}}{=} \neg wp(a, \neg Q')$ ,
- $prd_X(a) \stackrel{\text{def}}{=} wcp(a, x'_1 = x_1 \wedge \dots \wedge x'_n = x_n)$  that is a predicate on the state variables  $X = \{x_1, \dots, x_n\}$  in the source state before  $a$  and the target state variables  $X' = \{x'_1, \dots, x'_n\}$  after  $a$ .

## 2.2 Predicate Abstraction

Predicate abstraction [1] is a special instance of the framework of abstract interpretation [12] that maps the potentially infinite state space  $C$  of a concrete transition system onto the finite state space  $A$  of an abstract transition system via a set of  $n$  predicates  $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \dots, p_n\}$  over the state variables. The set of abstract states  $A$  contains  $2^n$  states. Each state is a tuple  $q \stackrel{\text{def}}{=} (q_1, q_2, \dots, q_n)$  with  $q_i$  being equal either to  $p_i$  or to  $\neg p_i$ , and we also consider  $q$  as the predicate  $\bigwedge_{i=1}^n q_i$ . We define a total abstraction function  $\alpha : C \rightarrow A$  such that  $\alpha(c)$  is an abstract state  $q$  where  $c$  satisfies  $q_i$  for all  $i \in 1..n$ . By a misuse of language, we say that  $c$  is in  $q$ , or that  $c$  is a concrete state of  $q$ .

Let us now define the abstract transitions as *may* ones. Consider two abstract states  $q$  and  $q'$  and an event  $e$ . There exists a *may* transition from  $q$  to  $q'$  by  $e$ , denoted by  $q \xrightarrow{e} q'$ , if and only if there exists at least one concrete transition  $c \xrightarrow{e} c'$  where  $c$  and  $c'$  are concrete states with  $\alpha(c) = q$  and  $\alpha(c') = q'$ .

We check predicate satisfiability thanks to SMT solvers. For a predicate  $P$ , we define the solver call  $SAT_c(P)$  as returning either a model of  $P$ , or **unsat** if  $P$  is unsatisfiable, or **unknown** if the solver failed to determine the satisfiability of  $P$ . We also define  $SAT(P)$  as the predicate that is true iff  $SAT_c(P)$  returns a model (showing that  $P$  is satisfiable). Let  $e \stackrel{\text{def}}{=} a$  be an event definition,  $q \xrightarrow{e} q'$  is a *may* transition iff  $SAT(wcp(a, q') \wedge q)$ . We compute a concrete witness  $c \xrightarrow{e} c'$  by using the before-after predicate:  $(c, c') := SAT_c(prd_X(a) \wedge q'[X'/X] \wedge q)$  where  $q'[X'/X]$  is the predicate  $q'$  in which each state variable  $x_i$  is substituted by  $x'_i$ .

## 2.3 May Transition Systems

Let us introduce *may* transition systems (MTS), which are transition systems with abstract states, and abstract *may* transitions. They are related to Modal Transition Systems [13–15], but with only the *may* modality.

**Definition 2 (May Transition System).** *Let  $\text{Ev}$  be a finite set of event names and  $\mathcal{P} \stackrel{\text{def}}{=} \{p_1, p_2, \dots, p_n\}$  be a set of predicates. Let  $A$  be a finite set of abstract states defined by  $\{p_1, \neg p_1\} \times \{p_2, \neg p_2\} \times \dots \times \{p_n, \neg p_n\}$ . A tuple  $\langle Q, Q_0, \Delta \rangle$  is an MTS if it satisfies the following conditions:*

- $Q (\subseteq A)$  is a finite set of states,
- $Q_0 (\subseteq Q)$  is a set of abstract initial states,
- $\Delta (\subseteq Q \times \text{Ev} \times Q)$  is a *may* labelled transition relation.

Now, Definition 3 associates an abstraction defined by an MTS with an event system.

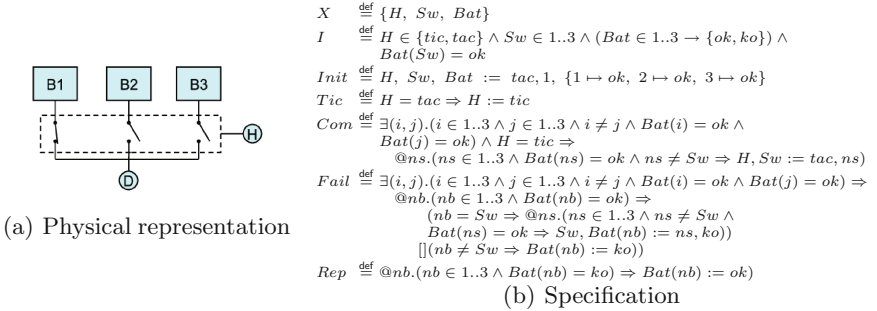
**Definition 3 (MTS associated with an ES).** Let  $ES \stackrel{def}{=} \langle X, I, Init, EvDef \rangle$  be an event system and  $\mathcal{P} \stackrel{def}{=} \{p_1, p_2, \dots, p_n\}$  be a set of  $n$  predicates over variables of  $X$  defining a set of  $2^n$  abstract states  $A \stackrel{def}{=} \{p_1, \neg p_1\} \times \{p_2, \neg p_2\} \times \dots \times \{p_n, \neg p_n\}$ . A tuple  $\langle Q, Q_0, \Delta \rangle$  is an MTS associated to  $ES$  and  $\mathcal{P}$  if it satisfies the following conditions:

- $Q \stackrel{def}{=} \{q \in A \mid \exists (q', e). (q \xrightarrow{e} q' \in \Delta \vee q' \xrightarrow{e} q \in \Delta)\}$ ,
- $Q_0 \stackrel{def}{=} \{q \mid q \in A \wedge (SAT(prd_X(Init) \wedge q[X'/X]))[X/X']\}$ ,
- $\Delta \stackrel{def}{=} \{q \xrightarrow{e} q' \mid q \in A \wedge q' \in A \wedge e \stackrel{def}{=} a \in EvDef \wedge SAT(wcp(a, q') \wedge q)\}$ .

Further in this paper, in Fig. 2 of Sect. 4, the reader will find an MTS example, whose ES is described in Fig. 1(b). The MTS is the part shown in dashed lines, with the four abstract states named  $q_0$  to  $q_3$  appearing as rounded rectangular dashed boxes. The abstract transitions of  $\Delta$  are represented as dashed arrows labelled by event names.

### 3 Illustrative Example: An Electrical System

To illustrate our approach, this section describes an electrical (**EL**) system example. It is a finite state control and command system that illustrates the MTS, as represented in Fig. 2.



**Fig. 1.** Electrical system and its specification

Figure 1(a) shows a device  $D$  powered *via* a switch to one of three batteries  $B_1, B_2$ , and  $B_3$ . A clock  $H$  periodically sends a signal that causes a commutation of the closed switch. The system has to meet the following requirements: one and only one switch is closed at a time, and a clock signal changes the switch that is closed. The batteries may break down. If it happens to the one that is powering

$D$ , an exceptional commutation is triggered. We assume that there is always at least one battery working. When there is only one battery working, the clock signals are ignored.

The event system in Fig. 1(b) uses three variables.  $H$  models the clock and takes two values:  $tic$  to ask for a commutation, and  $tac$  when it has occurred.  $Sw$  models the switches by indicating which one is closed.  $Bat$  models the batteries breakdowns by a total function that associates  $ok$  or  $ko$  (for a broken battery) to each battery. The state changes occur by applying four events:  $Tic$  sends a commutation signal,  $Com$  changes the closed switch responding to a  $Tic$ ,  $Fail$  breaks down at random a battery, and  $Rep$  repairs at random a broken battery.

The MTS (in dashed lines) of Fig. 2 in the next section abstracts the model of Fig. 1(b) w.r.t. the set of abstraction predicates  $\mathcal{P}_0 \stackrel{\text{def}}{=} \{p_1, p_2\}$ , where  $p_1 \stackrel{\text{def}}{=} H = tic$  (meaning that a commutation is asked) and  $p_2 \stackrel{\text{def}}{=} \exists(i, j).(i \in 1..3 \wedge j \in 1..3 \wedge i \neq j \wedge Bat(i) = ok \wedge Bat(j) = ok)$ .  $p_2$  means that at least two batteries are  $ok$ , so that a single battery is left working in the states where it is false.

## 4 Abstraction and Approximated Transition System Computation

This section presents an algorithm used to compute both an abstraction that is an MTS, and an under-approximation. The reunion of both is called an Approximated Transition System (ATS, defined in Definition 4), in which  $\langle C, C_0, \Delta^c \rangle$  is an under-approximation of the labelled transition system that is the semantics of the event system from which the MTS is deduced.

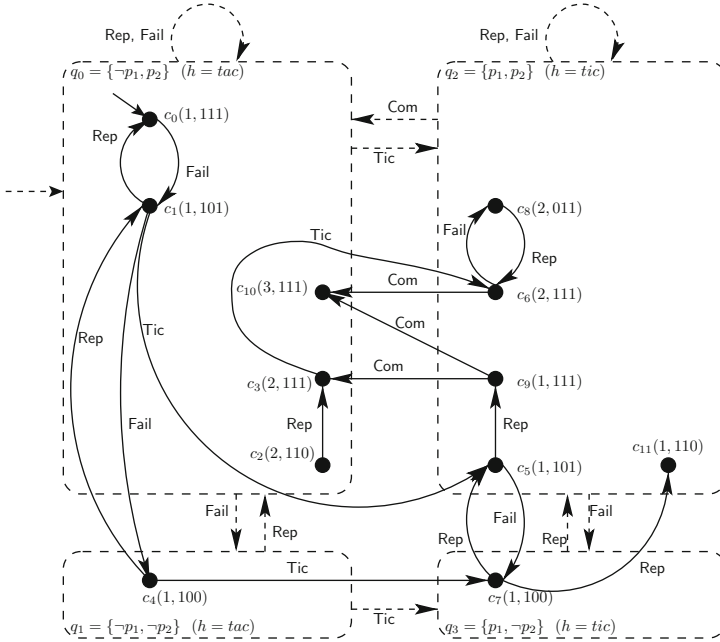
**Definition 4 (Approximated Transition System).** *Let  $\langle Q, Q_0, \Delta \rangle$  be an MTS. A tuple  $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c \rangle$  is an ATS whose  $\langle C, C_0, \alpha, \Delta^c \rangle$  is a concretization of the MTS  $\langle Q, Q_0, \Delta \rangle$  where:*

- $C, C_0$  are sets of respectively concrete states and concrete initial states,
- $\alpha$  is a total abstraction function from  $C$  to  $Q$ ,
- $\Delta^c (\subseteq C \times Ev \times C)$  is a concrete labelled transition relation.

For example, Fig. 2 shows an ATS of the electrical system of Fig. 1(a). The MTS appears in dashed lines while the full lines represent its concretization. The concrete states are showed as big dots.

In the rest of the paper, for the abstract states, we distinguish between the *may-reachability* and the *reachability*. The former is the reachability by the abstract *may* transition relation  $\Delta$ , and the latter is the reachability by the concrete transition relation  $\Delta^c$  in the ATS. We say that an abstract state  $q$  is reachable if there exists at least one concrete instance of  $q$  that is reachable thanks to the transition relation  $\Delta^c$ . By extension, an abstract transition is reachable if there exists at least one concrete instance in  $\Delta^c$  whose source state is reachable.

The ATS computation algorithm that we present concretizes the *may* transitions on the fly during the MTS computation. It guarantees that every *may*



**Fig. 2.** Example MTS and ATS of the Electrical System. The values of the concrete states are indicated in parentheses right by them. For example with state  $c_8$ , (2, 011) means that battery 2 is used, and that battery 1 is *ko* while batteries 2 and 3 are *ok*. The value of  $h$  is given globally in the abstract states.

transition between two abstract states is concretized. A total abstraction function  $\alpha$  maps each concrete state of  $C$  to an abstract state of  $Q$ . The algorithm comes in two versions, both of them being presented in the same figure. The first version is the one presented in this section. It is referred to as Algorithm 1. The second version, referred to as Algorithm 2, is enhanced by heuristics that are explained in Sect. 5. The differences between Algorithms 1 and 2 are highlighted and enclosed in square brackets. Read the left hand highlighted parts for Algorithm 1, and replace them with the right hand ones for Algorithm 2. Notice that since Algorithm 2 computes more things than Algorithm 1, some fictive empty parts (the empty highlighted square brackets  $\square$ ) have been added in Algorithm 1.

Lines 1 to 8 of Algorithm 1 compute the set of initial abstract states  $Q_0$ , an instance of each being recorded as a concrete witness in  $C_0$  with its association in  $\alpha$ . Lines 9 to 35 compute the *may* transition relation  $\Delta$ . Each abstract transition is concretized by a witness  $\{c_w \xrightarrow{e} c'_w\}$ , and the concrete states  $c_w$  and  $c'_w$  are recorded in  $C$  with their associations in  $\alpha$ . For that it computes in the set  $RQ$  the set of *may-reachable* states. For each *may-reachable* source state, it checks for each potential abstract state (line 12) and for each event (line 13) if a *may* transition exists (line 14). When it is the case, the algorithm records the witness

Algorithm: ATS computation		[1: without heuristics]	[2: with heuristics]
<b>Inputs</b>	: $\langle X, I, Init, EvDef \rangle$ : an Event System where $EvDef \stackrel{\text{def}}{=} \{e \stackrel{\text{def}}{=} a \mid e \in Ev\}$ A: a finite set of abstract states	[ ]	[orderStates: $2^A \times Q \rightarrow \text{list of Abstract States}$ (ordering function of the abstract states)]
<b>Output</b>	: $\langle Q, Q_0, \Delta, C, C_0, \alpha, \Delta^c, [ ] \rangle$ : an ATS [ ]	[ ]	[oEv: ordered list of the events of Ev] [ $\kappa$ ]
<b>Variables</b>	: $RQ$ : the set of abstract states remaining to be handled $q, q'$ : the source and target abstract states of the current transition $c, c'$ : the concrete source and target states of respectively $q$ and $q'$ $c_w, c'_w$ : the witness concrete source and target states of a <i>may</i> -transition $GC$ : the set of [already known] concrete states of $q$ [ ]	[ ]	[provided with a coloration function $\kappa \in C \rightarrow \{\text{green}, \text{blue}\}$ ] [green C-reachable]
		[ ]	[BC: the set of blue concrete states of $q$ ]
1	$Q := \emptyset; Q_0 := \emptyset; \Delta := \emptyset; C_0 := \emptyset; \alpha := \emptyset; \Delta^c := \emptyset; [ ]$	[ ]	[ $\kappa := \emptyset$ ]
2	<b>foreach</b> $q \in A$ <b>do</b>		
3	$c := (SAT_c(prd_X(Init) \wedge q[X'/X]))[X/X']$		
4	<b>if</b> $c \notin \{\text{unsat}, \text{unknown}\}$ <b>then</b>		
5	$Q_0 := Q_0 \cup \{q\}; C_0 := C_0 \cup \{c\}; \alpha(c) := q$		
6	[ ]		[ $\kappa(c) := \text{green}$ ]
7	<b>end</b>		
8	<b>end</b>		
9	$C := C_0; RQ := Q_0;$		
10	<b>while</b> $RQ \neq \emptyset$ <b>do</b>		
11	<b>choose</b> $q \in RQ; RQ := RQ - \{q\}; Q := Q \cup \{q\}$		
12	<b>foreach</b> $q' \in [A]$ <b>do</b>		[list orderStates(A, q)]
13	<b>foreach</b> $(e \stackrel{\text{def}}{=} a) \in [EvDef]$ <b>do</b>		[list oEv]
14	$(c_w, c'_w) := SAT_c(prd_X(a) \wedge q'[X'/X] \wedge q)$		
15	<b>if</b> $(c_w, c'_w) \notin \{\text{unsat}, \text{unknown}\}$ <b>then</b>		
16	$\Delta := \Delta \cup \{q \xrightarrow{e} q'\};$		
17	$GC := \{c_q \mid \alpha(c_q) = q\}$		[ $\wedge \kappa(c_q) = \text{green}$ ]
18	$(c, c') := SAT_c(prd_X(a) \wedge q'[X'/X] \wedge \bigvee_{c_q \in GC} c_q)$		
19	<b>if</b> $(c, c') \notin \{\text{unsat}, \text{unknown}\}$ <b>then</b>		
20	$C := C \cup \{c'\}; \alpha(c') := q'; \Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\};$		
21	[ ]		[ $\kappa(c') := \text{green}; BC := \{c'_q \mid \alpha(c'_q) = q' \wedge \kappa(c'_q) = \text{blue}\};$ ]
22	[ ]		[ $(c, c') := SAT_c(prd_X(a) \wedge (\bigvee_{c'_q \in BC} c'_q)[X'/X] \wedge \bigvee_{c_q \in GC} c_q)$ ]
23	[ ]		[ <b>if</b> $(c, c') \notin \{\text{unsat}, \text{unknown}\}$ <b>then</b>
24	$\Delta^c := \Delta^c \cup \{c \xrightarrow{e} c'\};$		
25	recursively colour in green from $c'$ ;		
26	<b>end</b>		
27	<b>end</b>		
28	$C := C \cup \{c_w, c'_w\}; \Delta^c := \Delta^c \cup \{c_w \xrightarrow{e} c'_w\}; \alpha(c_w) := q; \alpha(c'_w) := q';$		
29	[ ]		[ <b>if</b> $c_w \notin \text{domain}(\kappa)$ <b>then</b> $\kappa(c_w) := \text{blue}$ <b>end</b> ]
30	[ ]		[ <b>if</b> $c'_w \notin \text{domain}(\kappa) \vee \kappa(c_w) = \text{green}$ <b>then</b> $\kappa(c'_w) := \kappa(c_w)$ <b>end</b> ]
31	<b>if</b> $q' \notin Q$ <b>then</b> $RQ := RQ \cup \{q'\}$ <b>end</b>		
32	<b>end</b>		
33	<b>end</b>		
34	<b>end</b>		
35	<b>end</b>		

transition (see lines 16 and 28), but also possibly another concrete transition (see lines 17 to 27) whose source state is one of the existing concrete states of the current source state  $q$  when it exists. This last transition is computed first to improve the *reachability* of the concrete transition relation. Indeed, the existing concrete states are more likely to be connected to the initial states than the

witness source state provided by the solver in line 14. Last, line 31 adds  $q'$  as a *may-reachable* state that has not been taken into account yet to compute the *may* transition relation.

The algorithm terminates because it iterates on a finite number of abstract states and events. It is sound because the transitions computed are concrete instances of the semantics of the event system.

## 5 Heuristics for Better Abstraction Coverage

Using Algorithm 1, the connectivity and the reachability of the computed ATS might be weak, depending on which witnesses are exhibited by the solver. This section provides two heuristics, integrated into Algorithm 2, for improving both the connectivity and the reachability. The first heuristic addresses this problem by allowing the engineer to firstly define an order for the set of events of  $Ev$  in an ordered list  $oEv$  (line 13) and, secondly, a custom function ordering the set of abstract states  $A$  (line 12). The second heuristic, exposed in Sect. 5.2, adapts the partial computation of reachability proposed in [6] to our purpose for integrating it into Algorithm 2. The resulting new algorithm's complexity is the same as the previous one. The ATS of Fig. 2 was obtained by applying Algorithm 2 to the electrical system of Fig. 1(a), w.r.t. the set of predicates  $\mathcal{P}_0$  (defined in Sect. 3). The concrete states are numbered according to their order of discovery by Algorithm 2.

### 5.1 Events and States Ordering

Our first heuristic consists of providing means to control the order in which the events and abstract target states are handled by the algorithm.

Indeed, usually in reactive systems, some events can only be fired after other events have previously been executed. Let us consider the **EL** system where no battery repairing (modelled by the *Rep* event) can occur unless at least one battery has broken down first (modelled by the *Fail* event). Since Algorithm 1 currently uses an unordered set of events  $EvDef$ , it might attempt to concretize a *Rep* transition before trying to concretize any *Fail* transition. In this case, the concrete source state of the *Rep* transition would not be a reachable one. To fix this, we introduce the ordered list of events  $oEv$  as an input in Algorithm 2. To compute a complete abstraction, i.e. covering all events and all states,  $oEv$  must contain at least one occurrence of each event of the set  $EvDef$ . For example, for the **EL** system, in all judicious orders, *Fail* must precede *Rep* for the aforementioned reason, and *Tic* must precede *Com* because *Com* is a response to the event *Tic*.

Similarly, the *orderStates* function parameterizes Algorithm 2. Thanks to this function, the order in which the abstract target states are handled can be controlled. To compute a complete abstraction, the list returned by *orderStates* must contain at least all the abstract states of  $A$ . While being completely customizable by the engineer, the function used in our experiments presented in Sect. 6

gives better results for an order in which the first target abstract state handled is the source abstract state (state  $q$  in line 11) and the other states are ordered arbitrarily. Indeed, treating reflexive abstract transitions first tends to increase the number of reachable concrete states within the source abstract state. As a result, the chances that the next abstract transitions can be concretized from a reachable source state are increased.

When applying Algorithm 1 to the **EL** system with a set of abstraction predicates (first **AP** in Table 1), 33.33% of the abstract states and 11.11% (see line 1) of the abstract transitions are covered by the ATS. Integrating the event and state ordering without coloration improved these ratios respectively to 66.67% and 44.44%.

These ordering heuristics are integrated into Algorithm 2, along with the concrete states coloration discussed in Sect. 5.2. Even though our results did not focus on that point, an interesting perspective could be to consider the concretization of a same abstract transition multiple times. This could be useful for instance for systems requiring initialization steps that repetitively apply the same event, such as a credit card system for example. In fact, this behaviour can already be implemented using our algorithm by adding the same event to  $oEv$  several times, and by adding the target state of the abstract transition several times to the list returned by the *orderStates* function.

## 5.2 Concrete States Coloration

The reachability of the concrete states of the under-approximation is improved and computed on the fly in Algorithm 2 at no additional cost w.r.t. Algorithm 1. The principle is to associate a colour with each concrete state. A reachable state is coloured in green and a state whose reachability is unknown is coloured in blue. While Algorithm 1 tried to concretize the abstract transitions from an already known concrete state, Algorithm 2 uses the reachability information when concretizing the abstract transitions. It first tries to concretize an abstract transition from any known and *reachable* state (see lines 17 and 18). If it is indeed possible (line 19), the solver returns a first reachable concrete transition, added to the ATS, whose concrete target state becomes green (see lines 20 and 21). To improve the connectivity, the algorithm also tries to join a green source concrete state to a target blue one, whose *reachability* is thus currently unknown (line 22). If it is possible (line 23), its colour becomes green (line 24) since it is a target of a concrete transition starting from a *reachable* (green) concrete state. Even if the concretization from a known green concrete state is not possible, the abstract transition is still concretized. The corresponding concrete source and target states may already be known. In this case, their reachability remain the same. Otherwise, since we have no information about their reachability, they are coloured in blue (see lines 28 and 29).

When applying Algorithm 2 with coloration and without events and states ordering to the **EL** system with the first set of abstraction predicates in Table 1, 100% of the abstract states and 77.78% of the abstract transitions are covered by the ATS. This is better than with Algorithm 1 that gives respectively 33.33%

and 11.11%. Moreover, integrating the two heuristics seen in Sect. 5.1 into Algorithm 1 improved these two ratios to 100%.

These heuristics allow improving the reachability of the ATS for all the systems that we use in our experiments (see Sect. 6.2).

## 6 Implementation and Experimentation

This section introduces in Sect. 6.1 our proof-of-concept tool developed to evaluate the effects of the heuristics. The experimental results on four examples in Table 1 are presented in Sect. 6.2. Then, in Sect. 6.3 we analyse these results and conclude on the contributions of the heuristics presented in this paper.

### 6.1 About the Tool

The developed tool can be seen as a library for handling abstract and concrete transition systems as well as event systems. It embeds an event-B parser and allows to manipulate most event-B systems. The two algorithms are implemented and can be applied to them. The library also provides the user with many facilities for dealing with event systems. For instance, pre-implemented functions allow to easily compute an abstraction of a model from a set of abstraction predicates, as well as the *wp*, *wcp* and before-after predicates *prd<sub>X</sub>* of events defined by guarded actions. The library also contains functions to check the modality of abstract transitions and to find a concretization of an abstract state or an abstract transition. It can also be seen as a simple API for multiple SMT-solvers since the tool automatically generates SMT-Lib2 code for checking the satisfiability of any first order boolean formula. The tool is constituted of more than 5000 lines of JAVA code (version 8) and uses Z3 [16] as default SMT-solver. The library can be downloaded at <https://github.com/stratosphr/stratest/wiki>. This website also gives information on how to use the tool.

### 6.2 Experimental Results

This section provides the results obtained when applying Algorithm 1 and Algorithm 2 to a set of four realistic event systems of increasing size. These event systems, available in the aforementioned GitHub repository, were taken back from various previous work without modification so as not to influence the experiment and threaten the validity of the results. The set of examples contains the electrical system (**EL**) presented in Sect. 3, a phone book service (**PH**), a coffee machine system (**CM**), and a car alarm system (**CA**). For each of them, two different sets of abstraction predicates have been used (see the **AP** column).

The following column names appear in Table 1: **Sys** for the system studied and an upper approximation of its size between parentheses, **#Ev** for the number of events in the event system, **AP** for an identification of the set of abstraction predicates used, **#AP** for the number of abstraction predicates in **AP**, **Alg.** for the algorithm applied, **#AS** for the number of *may-reachable* abstract



states,  $\#AS_{reach}$  for the number of *reachable* abstract states computed,  $\tau_{AS}$  for the abstract state coverage that is the ratio  $\frac{\#AS_{reach}}{\#AS}$ ,  $\#AT$  for the number of abstract transitions,  $\#AT_{reach}$  for the number of *reachable* abstract transitions computed. Note that we say that an abstract transition is *reachable* if there exists a concrete instance of it in the ATS that is reachable. Next, there are the following column names:  $\tau_{AT}$  for the abstract transition coverage that is the ratio  $\frac{\#AT_{reach}}{\#AT}$ ,  $\#CT$  for the number of concrete transitions computed,  $\rho_{CT}$  for the ratio  $\frac{\#CT}{\#AT_{reach}}$  which measures the efficiency of the method, by indicating in average how many concrete transitions have been computed for making an abstract transition reachable, and finally **Time** for the ATS computation runtime (in seconds). The connectivity between transitions is indirectly measured via the coverage and efficiency rates, since a reachable state or transition is necessarily *connected* to a concrete initial state.

The main results of our method are the coverage ratios of abstract states ( $\tau_{AS}$ ) and abstract transitions ( $\tau_{AT}$ ). For almost identical computation time(s), an improvement of these ratios indicates a better performance of the method. For  $\rho_{CT}$ , a value between 3 and 1 indicates that the algorithm covers one abstract transition per iteration step. When this ratio decreases that indicates an improvement of the efficiency. Indeed, for each abstract transition, each iteration step computes one up to three transitions according to the conditions in lines 19 and 23. For the **EL** system, with the first set of abstraction predicates,  $\rho_{CT}$  decreases from 13 to 2, meaning that the heuristic allowed to compute more interesting concrete transitions, increasing the abstraction transition coverage from 11.11% to 100%.

### 6.3 Analysis of the Obtained Results

This section comments on the results exposed in Table 1.

**Table 1.** ATS computation results

Sys	#Ev	AP	#AP	Alg.	#AS	#AS <sub>reach</sub>	$\tau_{AS}$ (%)	#AT	#AT <sub>reach</sub>	$\tau_{AT}$ (%)	#CT	$\rho_{CT}$	Time
EL (24)	4	1	2	1	3	1	33.33	9	1	11.11	13	13	00.283
				2	3	3	100	9	9	100	18	2	00.297
		2	2	1	4	4	100	11	8	72.73	15	1.88	00.429
				2	4	4	100	11	11	100	17	1.55	00.449
PH (2 <sup>10</sup> )	4	1	3	1	3	3	100	12	11	91.67	16	1.45	00.261
				2	3	3	100	12	12	100	22	1.83	00.287
		2	6	1	8	8	100	62	60	96.77	83	1.38	01.994
				2	8	8	100	62	62	100	88	1.42	02.204
CM (2 <sup>16</sup> )	8	1	3	1	4	3	75	30	5	16.67	47	9.4	00.753
				2	4	4	100	30	24	80	54	2.25	00.854
		2	3	1	6	3	50	52	7	13.46	83	11.86	01.639
				2	6	6	100	52	25	48.08	77	3.08	01.629
CA (2 <sup>15</sup> )	20	1	6	1	8	5	62.5	31	18	58.065	44	2.44	13.818
				2	8	8	100	31	25	80.65	50	2	13.283
		2	9	1	9	5	55.56	30	11	36.67	37	3.36	23.978
				2	9	9	100	30	28	93.33	52	1.86	25.381

As expected, the ATS computation times are nearly identical, no matter which version of the algorithm has been used. Note that for two cases out of eight the ATS computation time with Algorithm 2 is on average slightly faster than with Algorithm 1. Since the formulas whose satisfiability is checked are different between the two algorithms, the solver may be faster to provide an answer for the formulas in Algorithm 2 than in Algorithm 1.

We observe that Algorithm 2 improves both the abstraction coverage rates and the efficiency  $\rho_{CT}$  compared to Algorithm 1. In particular, we point out the **CM** case with the first **AP** where the transition coverage and the efficiency achieved by Algorithm 2 is about respectively five and four times better than by Algorithm 1.

For all systems and all sets of abstraction predicates, the abstraction coverage is improved by Algorithm 2. Depending on the set of predicates used, the coverage for states and transitions can reach up to 100%. On most examples however, Algorithm 1 covers less than half of the abstract states and transitions. Note for example the **CM** case with the second set of abstraction predicates where the abstract states and transitions coverage(s) are respectively twice and three times better using the heuristics. All these results empirically confirm the interest of the proposed heuristics to improve the abstraction coverage.

The heuristics also produced good results concerning the efficiency rate  $\rho_{CT}$ . For most cases, its value is decreased by Algorithm 2 thanks to the heuristics, which means that they generally help concretizing abstract transitions by useful transitions improving the abstraction coverage. For the **CM** system with the second **AP** and Algorithm 1 for example, an average of 11.86 concrete transitions need to be computed in order to cover one abstract transition. When the heuristics are used however, an average of only 3.08 concrete transitions computation is needed to cover one abstract transition.

The ordering heuristic alone does not necessarily improve the abstraction coverage w.r.t. Algorithm 1, whereas the coloration heuristic alone always improves the results. Nevertheless, for four cases out of the eight presented in this paper, combining the ordering and coloration heuristics improves the abstraction coverage compared to coloration only. For the **CM** system with the first **AP** for example, the ordering heuristic alone covers two abstract states compared to three with Algorithm 1, and six abstract transitions compared to five. The coloration heuristics alone covered all of the four abstract states, and twenty-two out of the thirty abstract transitions. When combining both heuristics, the four abstract states and twenty-four abstract transitions are covered.

## 7 Related Work

In [17,18], the set of abstraction predicates is iteratively refined in order to compute a bisimulation of the semantics of the model when it exists. None of these two methods is guaranteed to terminate, because of the refinement step that sometimes needs to be repeated endlessly. SYNERGY [19] and DASH [20] combine under-approximation and over-approximation computations to check

safety properties on programs. As we aim at proposing an efficient method to build a *reachable* under-approximation of a system that covers all abstract states and all abstract transitions w.r.t. a specification and a set of predicates, our algorithm does not refine the approximation and so always terminates.

The closest methods to ours are those that are proposed in [6,21]. These approaches propose algorithms that compute an under-approximated concretization of a predicate abstraction covering its abstract states and transitions. Both these methods are exploited for generating tests. The algorithm in [21] does not traverse nor compute the *may* abstraction. It builds a partial concretization of the abstract states that are *reached* from an initial concrete state by a forward walk. To improve this method, the algorithm in [6] computes exhaustively the *may* abstraction by random abstract state generation. Therefore, some generated concrete states are not *reached*. Then Veanes and Yavorsky [6] propose to distinguish between four kinds of abstract transitions: green transitions when there exists an instance that is reached from an initial concrete state, blue transitions when there exists instances, but none known to be reachable from an initial state, red transitions when there does not exist any instance, and grey transitions for the transitions that have not been concretized yet. In our method, we compute and concretize only the part of the *may* abstraction that is *may-reachable* by an abstract transition from an initial abstract state. We do not record the red transitions that are non-existing transitions in the MTS, and we do not need the grey transitions that are the ones which remain to be treated. In contrast with [6], our method colours the concrete states instead of the abstract transitions. This allows us to distinguish between the *reached* states (green) and the states for which we do not know whether they are *reached* (blue) or not. So for improving the method, Algorithm 2 connects in priority a green source state  $s$  to a blue target state  $t$ . That has a “domino effect” because all the blue reached states from  $t$  remain blue, but become *reachable*.

Some other work under-approximate an abstraction for generating tests. The tools Agatha [22], DART [23], CUTE [24], EXE [25] and PEX [26] also compute abstractions from models or programs, but only by means of symbolic executions [27]. This data abstraction approach computes an execution graph. Its set of abstract states is possibly infinite whereas it is finite with our method.

Our method can be applied to generate tests as the concolic execution in [24]. The concolic method allows to generate structural tests of systems covering partially the control flow that must be explicit. Our approach allows to generate tests covering the paths defined by the set of abstraction predicates for systems whose control flow is implicitly defined.

## 8 Conclusion and Further Work

This paper has presented an algorithmic method for computing a concrete approximation of the predicate abstraction of an event system. All of the abstract states and transitions are covered, but as the control flow is implicit in an event system, our method focuses on computing concrete sequences that are connected

and reachable. We have presented two heuristics allowing us to better reach and connect these sequences. One heuristic colours the states that are known to be reachable, and the other takes a user defined order on the events and abstract states enumeration as parameters. Experimental results on four case studies are exhibited to confirm the practical interest of our approach.

As future work, we intend to define other means for guiding the sequences instantiation, in addition to the events ordering. We could introduce a relevance function on concrete states, as is done in [21], for targeting peculiar concrete states considered as more relevant. Also, our intention is to use the concrete sequences that we compute as model-based tests issued from a formal model of the specification. Abstracting this model would allow selection criteria such as paths selection to be used, when the size of the explicit model would prevent it.

## References

1. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). [https://doi.org/10.1007/3-540-63166-6\\_10](https://doi.org/10.1007/3-540-63166-6_10)
2. Godefroid, P., Jagadeesan, R.: On the expressiveness of 3-valued models. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 206–222. Springer, Heidelberg (2003). [https://doi.org/10.1007/3-540-36384-X\\_18](https://doi.org/10.1007/3-540-36384-X_18)
3. Abrial, J.R.: Modeling in Event-B: System and Software Design. Cambridge University Press, Cambridge (2010)
4. Dijkstra, E.: Guarded commands, nondeterminacy, and formal derivation of programs. *Commun. ACM* **18**(8), 453–457 (1975)
5. Dijkstra, E.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
6. Veanes, M., Yavorsky, R.: Combined algorithm for approximating a finite state abstraction of a large system. In: ICSE 2003/Scenarios Workshop, pp. 86–91 (2003)
7. Abrial, J.R.: The B Book. Cambridge University Press, Cambridge (1996)
8. Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.): ASM 2000. LNCS, vol. 1912. Springer, Heidelberg (2000). <https://doi.org/10.1007/3-540-44518-8>
9. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Log.* **1**(1), 77–111 (2000)
10. Bert, D., Cave, F.: Construction of finite labelled transition systems from B abstract systems. In: Grieskamp, W., Santen, T., Stoddart, B. (eds.) IFM 2000. LNCS, vol. 1945, pp. 235–254. Springer, Heidelberg (2000). [https://doi.org/10.1007/3-540-40911-4\\_14](https://doi.org/10.1007/3-540-40911-4_14)
11. Bride, H., Julliand, J., Masson, P.A.: Tri-modal under-approximation for test generation. *Sci. Comput. Program.* **132**(P2), 190–208 (2016)
12. Cousot, P., Cousot, R.: Abstract interpretation frameworks. *J. Log. Comput.* **2**(4), 511–547 (1992)
13. Larsen, K.G., Thomsen, B.: A modal process logic. In: LICS, pp. 203–210 (1988)
14. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-44685-0\\_29](https://doi.org/10.1007/3-540-44685-0_29)

15. Ball, T.: A Theory of predicate-complete test coverage and generation. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2004. LNCS, vol. 3657, pp. 1–22. Springer, Heidelberg (2005). [https://doi.org/10.1007/11561163\\_1](https://doi.org/10.1007/11561163_1)
16. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-78800-3\\_24](https://doi.org/10.1007/978-3-540-78800-3_24)
17. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 435–449. Springer, Heidelberg (2000). [https://doi.org/10.1007/10722167\\_33](https://doi.org/10.1007/10722167_33)
18. Păsăreanu, C.S., Pelánek, R., Visser, W.: Predicate abstraction with under-approximation refinement. *LMCS* **3**(1:5), 1–22 (2007)
19. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: a new algorithm for property checking. In: SIGSOFT FSE, pp. 117–127 (2006)
20. Beckman, N.E., Nori, A.V., Rajamani, S.K., Simmons, R.J., Tetali, S., Thakur, A.V.: Proofs from tests. *IEEE Trans. Software Eng.* **36**(4), 495–508 (2010)
21. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA, pp. 112–122 (2002)
22. Rapin, N., Gaston, C., Lapitre, A., Gallois, J.P.: Behavioral unfolding of formal specifications based on communicating extended automata. In: ATVA (2003)
23. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223 (2005)
24. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/SIGSOFT FSE, pp. 263–272 (2005)
25. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: ACM CCS, pp. 322–335 (2006)
26. Tillmann, N., de Halleux, J.: Pex–white box test generation for.NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-79124-9\\_10](https://doi.org/10.1007/978-3-540-79124-9_10)
27. Păsăreanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. *STTT* **11**(4), 339–353 (2009)

# Implementing the Symbolic Method of Verification in the C-Light Project

Dmitry Kondratyev(✉)

Institute of Informatics Systems SB RAS, Novosibirsk 630090, Russia  
apple-66@mail.ru  
<http://persons.iis.nsk.su/ru/postgraduates/kondratyev>

**Abstract.** The C-light is a project aimed onto deductive verification of C programs. It relies on three basic ideas, namely—metageneration of verification conditions (MetaVCG), semantic mark-up and the symbolic method, two-level verification that uses C-light language as a front-end and C-kernel language as a back-end. The semantic mark-up extends the standard Hoare inference rules by semantic labels for explanations of failed verification conditions. The symbolic method is based on a replacement of each for-loop by a single assignment with a cumulative effect, it allows us to avoid explicit invariant generation. However, to make verification efficient, it is necessary to develop new techniques instead of the replacement. These new techniques for verification of linear algebra programs is presented and explained in this article.

**Keywords:** MetaVCG approach · Semantic mark-up method  
Symbolic method of verification of definite iterations

## 1 Introduction

C-light verification project [13] is under development in the Laboratory for theory of programming, Institute of Informatics Systems (SB RAS). The C-light language is a representative subset of the C language [12]. One of the goals of the project is the development of an extendable self-applicable verification condition generator (VCG) for the C language.

A formal semantics of a programming language is one of prerequisites for deductive program verification. A so-called operational semantics is very natural to understand how programs work because it defines some abstract machine to execute programs. (By the way, since of C-light abstract machine has un-structured memory model, C-light doesn't support machine word level operations.)

However, operational approach is commonly used for model checking; instead in deductive verification a so-called axiomatic semantics is in use. On the

---

This research was in part supported by RFBR (grant No. 15-01-05974 and grant No. 17-01-00789).

other hand, the axiomatic approach may be too complicated for language like C-light [12], and so a two-level method of deductive verification has been employed. In the first stage C-light is translated into the intermediate language C-kernel [13]. This stage is necessary for elimination of constructs that are complicated for axiomatic semantics. A set of formal rules is used for this translation. For example, increment operators are eliminated by translation into pieces of code with assignments and addition. The C-kernel language is a strongly limited subset of the C-light language [11]. In the second stage verification conditions (VCs) are generated for the intermediate C-kernel program. Then generated verification conditions are passed to the theorem prover. The verification condition generation is based on the axiomatic semantics of C-kernel [11].

Two new objectives result from the logic of the C-light project [6]. The first objective is to develop and implement a method, which allows us to extend the C-kernel language by new program constructs without major changes in verification condition generator. The second objective is to develop highly specialized domain-oriented verification condition generators based on domain-specific methods for specialized classes of programs. It has been decided to use the concept of meta verification condition generation (MetaVCG) to address these objectives. Initially this concept was proposed and developed by Moriconi and Schwartz [9].

A conventional verification condition generator is a stand-alone highly specialized program. However, MetaVCG is an entirely different approach to creating a verification condition generator. The meta verification condition generator takes axiomatic semantics (in a special format) and automatically produces a VCG. The correctness and completeness of MetaVCG are ensured by using as an input only axiomatic semantics in very special format. Consequently, it is necessary to validate whether axiomatic semantics meets limitations before using MetaVCG.

The following problems may occur in the practical application of deductive verification: the program may be incorrect, specifications may be incorrect, the automatic theorem prover may be inefficient, the underlying theory may be incomplete. Once any of these happens, the user of the verification system will receive just a set of unproven VCs without any information about the reasons of the failure.

The metageneration idea allows the C-light verification system to be supplemented with the semantic mark-up method. The method focuses on such problems as the analysis, tracing and explanation of VCs. This method was proposed by Denney and Fisher [3]. It is based on the extension of the Hoare rules by semantic labels that can be used to generate explanations (in plain language) of the VCs.

The problem of automatic generation of loop invariants is undecidable [8]. However, the symbolic method of verifying for-loops with so-called definite iterations [10] allows us to tackle this problem. This opportunity is based on applying special inference rules to such loops. These rules allow us to avoid invariants. They are based on a special replacement function that represents the

commutative loop effect in a symbolic form. The MetaVCG approach allows the C-light system to be easily supplemented with such inference rules. The paper presents experimental application of MetaVCG, semantic labels, and symbolic method to verification of C-light program from linear algebra domain.

The paper is organized as follows. Section 2 presents theoretical basis of the research, namely: a brief introduction to deductive program verification, the MetaVCG approach, and the semantic mark-up method. Section 3 provides details of symbolic method of verifying definite iterations with a special attention to replacement operation. Section 4 addresses linear algebra program verification with a particular attention to inference rule for matrix column update. Verification case study is described in the Sect. 5 where program that calculates dot-product is verified on base of symbolic method and with aid of PVS verification system. A summary of results and plans for the future research are presented in the concluding Sect. 6.

## 2 Theoretical Background

This section sketches basics of deductive program verification, the MetaVCG approach and the semantic mark-up method.

### 2.1 Deductive Program Verification

Deductive program verification is applied to the Hoare triple [1]. The first part of a Hoare triple is a precondition. It is a logical formula. The second part of a Hoare triple is a program fragment. It is a list of statements. The third part of a Hoare triple is a postcondition. It is a logical formula also. In a standard notation  $\{\mathcal{P}\} S \{\mathcal{Q}\}$  for Hoare triples  $\mathcal{P}$  is the precondition,  $S$  is the program fragment,  $\mathcal{Q}$  is the postcondition. Given the Hoare triple, the precondition and the postcondition specify desirable behavior of the program fragment. Therefore, they are referred to as specifications or annotations, and the Hoare triple is referred to as the annotated program fragment.

The deductive program verification is automatic derivation of valid (partially correct) Hoare triples. The partial correctness of the Hoare triple means that if the precondition is true before the execution of the program fragment and if its execution terminates, then the postcondition is true upon its completion [9].

The inference rule has the following structure:

$$\frac{\psi_1, \dots, \psi_n}{\varphi} \quad (1)$$

where  $\psi_1, \dots, \psi_n$  are the premises (Hoare triples and logical formulas) and  $\varphi$  is the conclusion (Hoare triple). This notation means that  $\varphi$  is derived from  $\psi_1, \dots, \psi_n$ . The semantics of simple program statements (for example, the assignment operator) is normally based on a set of axioms, and the semantics of any compound statement is often based on an inference rule. The syntax-driven axiomatic system (i.e. that contains inference rules for all syntax constructs of the programming language) is called Hoare logic or axiomatic semantics.



Verification conditions generator (VCG) is a procedure, which reduces derivation the annotated program fragments to validation of program-free logical formulas. These formulas are referred to as verification conditions. Since programs are written in particular programming languages, a VCG depends on a particular programming language. Furthermore, a VCG is based on the axiomatic semantics of a particular programming language. Consequently, it allows a VCG to derive VCs from the Hoare triple.

The weakest precondition (wp) method is one particular VCG. Given the Hoare triple  $\{P\} S \{Q\}$ , the wp is denoted by  $wp(S, Q)$ . The wp is the precondition that ensures the truth of the corresponding Hoare triple and it can be derived from other preconditions that guarantees the truth of the corresponding Hoare triple. The Hoare triple is valid if and only if  $P$  implies  $wp(S, Q)$ . Thus, VCG can be implemented as a program that calculates  $wp(S, Q)$  [9].

As an example, let us consider the inference rule from a very popular textbook [1]:

$$\frac{\{P \wedge B\} S \{P\}, \quad P \wedge \neg B \supset Q}{\{P\} \text{ while } (B) \text{ assert } P \text{ do } S \{Q\}} \quad (2)$$

It is necessary to use induction to derive the Hoare triple for the **while** loop. Induction statement in this case is called the loop invariant: it is a statement that is true before loop execution, true after each loop iteration, and that ensures the correctness of loop exit.

## 2.2 The Extensions of the VCG

In academic settings it is possible to assume that a programming language isn't evolving and hence VCG shouldn't evolve also. But this static assumption is not true in industrial settings.

Firstly, the programming language may be supplemented with new language constructs. For example, plans of extending the C language are considered [6]. The new language constructs that appear in the new C language standard (C11) are of great interest. Also, C-light [13] can be extended by language constructs of related languages such as Objective C and C++.

Secondly, the implementation of highly specialized versions of the VCG is of interest. They should be based on specific methods of applying inference rules and classes of programs. This specialization makes verification simpler. Let us consider an example.

This example relates to linear algebra [14]. The axiomatic semantics of the class of linear algebra programs has been developed in the Laboratory for theory of programming [16]. The class is for vector and matrix manipulations. Given  $M$  is a rectangle matrix and an expression  $e = e(k, i)$  that depends on the indices  $k$  and  $i$ , let us consider the following template [14]:

```
{Q(M ← rep(M, mat(e1, e2, e3, e4), e(s, t)))}
  for(k = e1; k <= e2; k++)
    for(i = e3; i <= e4; i++)
      M[k][i] = e(k, i);
{Q}
```

where the matrix  $rep(M, mat(e_1, e_2, e_3, e_4), e(s, t))$  results from the replacement of all elements of the submatrix  $mat(e_1, e_2, e_3, e_4)$  by the expression  $e$ . Note that the problem of automatic generation of loop invariants is an algorithmically undecidable. However, the symbolic method of verifying for-loops allows us to avoid invariant generation by using some special constructs ( $rep$  and  $mat$ ). However, it leads to necessity to modify the VCG to implement this approach.

Thus, an extendable VCG is a necessity. A non-specialized VCG that contains all possible axioms and inference rules is not a good option. Instead a set of specialized VCG is a better solution. Consequently, approaches that make the implementation of such sets simpler are of great interest. Due to these reasons the MetaVCG approach [9] is implemented in the C-light project.

A VCG is based on the axiomatic semantics of a particular programming language. The metagenerator takes axiomatic semantics in the normal form and automatically produces a VCG. The wp method is used for building the VCG. The axiomatic semantics should be given in the normal form to guarantee the correctness and completeness of MetaVCG [9]. The normal form imposes rigid syntax constraints on axiomatic semantics. Many classical inference rules do not satisfy these constraints, so some preliminary transformation to equivalent form is required [6].

Note that it is necessary to generate explanations of the VC in the case of verification failure. Thus, the VCG must supplement the VC with special information. In the semantic mark-up method, this information appears in the form of semantic labels. With this approach [3], natural language explanations of the VC are generated. Consequently, the VCG must add labels to a special position in the VC. The VCG should use modified Hoare rules to do that. The modified Hoare rules are based on a special semantic mark-up (i.e., label types and positions) [6]. Note that term  $term$  supplemented by label  $label$  is denoted by  $\lceil term \rceil^{label}$ . Labels are added to the following three places: the incoming postcondition of a recursive VCG call in the premise of an inference rule, to the wp, or to the generated VC.

Note that MetaVCG takes inference rules in the normal form as an input. They define patterns of program constructs. The classical graphical representation of inference rules is easy to read, but is hard to input using a keyboard. Thus, the important task is to develop a language for defining them. This language should be based on first-order logic and the target language [6]. However, inference rules define the semantics of specific programs. Thus, a programming language has to be supplemented with some metaidentifiers. More detailed information about the C-light project can be found in [6, 11, 12].

### 3 The Essence of Symbolic Method

Let us consider a so-called definite iteration over data structures [10]. The body of a loop for definite iteration is a sequence of assignment operators and conditional operators. It can be represented by the vector assignment operator  $v = body(v, x)$ , where  $x$  is the iteration parameter,  $v$  is the vector of other

variables, and  $body(v, x)$  is the vector of conditional expressions that is based on *if-then-else* operation. This form may result from applying a sequence of appropriate substitutions that replace conditional operators by conditional expressions and replace the sequence of assignment operators by an assignment operator.

Let us consider the form of definite iteration over data structures:

$$\text{for } x \text{ in } S \text{ do } v = body(v, x) \text{ end} \quad (3)$$

where  $S$  is a structure (possibly hierarchical). The initial value of the vector  $v$  is  $v_0$ . If  $empty(S)$  is true, then the result of the iteration is  $v_0$ . If  $\neg empty(S)$  and  $vec(S) = [s_1, \dots, s_n]$ , then the loop body is executed sequentially for  $x$  that takes on a value from  $s_1, \dots, s_n$ .

Let us introduce some notation. The result of the substitution of the expression  $exp$  for all occurrences of the variable  $y$  in the formula  $R$  is denoted by  $R(y \leftarrow exp)$ . The result of the simultaneous substitution of the expressions of the vector  $vecp$  for all occurrences of the variable  $vec$  in the formula  $R$  is denoted by  $R(vec \leftarrow vecp)$ . The inference rule for iteration (3) is based on the replacement operation  $rep(v, S, body)$ , where the function represented by the loop body is denoted by  $body$ . The replacement operation  $rep(v, S, body)$  represents the loop effect in a symbolic form. If  $v_0 = v$ ,  $n = 0$  and  $empty(S)$ , then  $rep(v, S, body) = v_n$ . If  $i = 1, \dots, n$ ,  $\neg empty(S)$  and  $vec(S) = [s_1, \dots, s_n]$ , then  $v_i = body(v_{i-1}, s_i)$ . The following theorem describes useful features [10] of the replacement operation:

**Theorem 1.** *Iteration (3) is equal to the vector assignment  $v = rep(v, S, body)$ .*

The following inference rule [10] results from Theorem 1:

$$\frac{\{P\} prog \{Q(v \leftarrow rep(v, S, body))\}}{\{P\} prog; \text{for } x \text{ in } S \text{ do } v = body(v, x) \text{ end} \{Q\}} \quad (4)$$

where  $P$  is the precondition,  $Q$  is the postcondition that is independent of the iteration parameter,  $prog$  is the program fragment, the partial correctness of the program  $prog$  relative to  $P$  and  $Q$  is denoted by  $\{P\} prog \{Q\}$ .

The VCs that are based on using the replacement operation result from the application of this rule. The induction or problem-oriented approach is used to prove such VCs.

The MetaVCG [9] allows this rule to be easily introduced to the C-light system. It satisfies the normal form constraints. Consequently, this rule can be used as an argument of the metagenerator. The language of defining axioms and inference rules can be used to define it. Let us consider this form of it:

```
{P} prog {substitution(Q, v, rep(v, S, body))},
|- {any_predicate(P)} any_code(prog) for_iteration(x, v, S, body)
{any_predicate(Q)}
```

where the substitution of  $v$  for  $rep$  is denoted by  $substitution(Q, v, rep(v, S, body))$  and the definite iteration is denoted by  $for\_iteration(x, v, S, body)$ .

Note that `for_iteration(...)` is the template that will be matched with program constructs by the metagenerator. The implementation of the matching algorithm in the metagenerator allows this template to be used. This implementation allows definite iteration over arrays to be matched with this template at the current stage of work. The extension of this algorithm by matching definite iteration over other structures such as trees with this template is a further research topic.

Generation of *rep* operation is based on symbolic computation of variables from vector *v*. Arithmetic operations, logic operations, conditional operator, subscript operator and assignment can be processed by the *rep* generator at the current stage. Note that conditional operator and assignment are base operations of the symbolic method of verification of definite iterations. We plan to extend our implementation by other operations.

MetaVCG allows the semantic mark-up method to be implemented. Thus, this rule has been supplemented with semantic labels. Let us consider them:

$$\frac{\{P\} \text{ prog } \{ \lceil Q(v \leftarrow \lceil \text{rep}(v, S, \text{body}) \rceil^{\text{rep\_iter}} \rceil^{\text{for\_iter}} \} \}}{\{P\} \text{ prog}; \text{ for } x \text{ in } S \text{ do } v = \text{body}(v, x) \text{ end } \{Q\}} \quad (5)$$

The MetaVCG approach allows us to avoid modifying the C-light system “manually” to add new concepts of semantic labels. Thus, two new concepts of labels have been implemented in the C-light project easily. The subformula that results from the replacement operation has been supplemented with the *rep\_iter* concept. The postcondition that has been changed by substitution has been supplemented with the *for\_iter* concept. The language of defining axioms and inference rules has been supplemented with new language constructs for semantic labels. Let us consider it:

```
{P} prog {(label for_iter substitution(Q, v,
                                (label rep_iter rep(v, S, body))))}
|- {any_predicate(P)} any_code(prog) for_iteration(x, v, S, body)
{any_predicate(Q)}
```

The MetaVCG approach allows the C-light project to be supplemented with the symbolic method of verifying definite iterations.

## 4 Towards Verification of Linear Algebra Programs

The elimination of loop invariants using the symbolic method may be applied to a wide class of programs. Note that the axiomatization of some special forms of the replacement operation has already been created [16]. But in the present paper we consider only verification of linear algebra programs.

Linear algebra programs are based on special for-loops [15]. These loops permute matrix rows, or divide a row by a number, or subtract a row multiplied by a number from another row [14]. The invariants of such loops can be represented by the replacement operation. This operation allows special inference rules to be defined without using loop invariants. However, it is necessary to check special conditions to apply such rules.

Let us consider an example of such a rule [14]. The matrix that has been created from  $M$  by the replacement of each matrix element  $(s, t) \in S$  (where  $S$  is a set of matrix indices) by the value of the expression  $e(s, t)$  is denoted by  $rep(M, S, e(s, t))$ . Thus, the following rule is based on this replacement operation:

$$\frac{\{P\} \text{ prog } \{Q(M \leftarrow rep(M, col(i, e_1, e_2), e(s, t)))\}}{\{P\} \text{ prog; for}(k = e_1; k \leq e_2; k++) \quad M[k][i] = e(k, i) \quad \{Q\}} \quad (6)$$

This rule represents the replacement of a part of a matrix column by the expression that depends only on this column and the current row. The set that includes each matrix element of the column  $l$  from the row  $m$  to row  $n$  is denoted by  $col(l, m, n)$ . Consequently,  $(u, v) \in col(l, m, n) \iff (v = l \wedge m \leq u \leq n)$ . There is the following condition of applying this inference rule: *if  $e(i, k)$  depends on  $M[m][n]$  then  $(m, n) \notin col(i, e_1, k - 1)$* . Let us consider multiple conditions of applying inference rules for linear algebra programs [16]. These conditions ensure the independence of the replacement operation  $rep(M, S, e(s, t))$  from the order of selecting elements of the set of matrix indices  $S$ .

This rule satisfies normal form constraints [9]. Thus, it can be used as an argument of the metagenerator. The language of defining axioms and inference rules can be used to define it. Let us consider such a definition of this rule:

```
{P} prog {substitution(Q, M, rep(M, col(i, e1, e2), e(s, t)))},
|- {any_predicate(P)} any_code(prog) for_col(M, i, k, e1, e2, e(s, t))
{any_predicate(Q)}
```

where the replacement of  $M$  by  $rep(M, col(i, e_1, e_2), e(s, t))$  in  $Q$  is denoted by  $substitution(Q, M, \dots)$ , the template that corresponds to iteration over the matrix column is denoted by  $for\_col(M, i, k, e_1, e_2, e(s, t))$ . The algorithm of matching templates and a program construct that has been implemented in the metagenerator allows this template to be compared successfully with the loop body. Also, this algorithm checks the condition of applying this rule. There were no for-loops in the C-kernel language. Thus, this algorithm allows a special form of for-loop to be introduced easily to this language.

MetaVCG allows the semantic mark-up method to be implemented in the C-light system. Let us consider this rule as supplemented with semantic labels:

$$\frac{\{P\} \text{ prog } \{[Q(M \leftarrow [rep(M, col(i, e_1, e_2), e(s, t))]^{rep\_col})]^{for\_col}\}}{\{P\} \text{ prog; for}(k = e_1; k \leq e_2; k++) \quad M[k][i] = e(k, i) \quad \{Q\}} \quad (7)$$

The MetaVCG approach allows the C-light project to be extended by new concepts of semantic labels. Thus, two new concepts of labels have been easily implemented in the C-light project. The subformula that results from the replacement operation has been supplemented with the *rep\_col* concept. The postcondition that has been changed by substitution has been supplemented with the *for\_col* concept. The language of defining axioms and inference rules

allows expressing this inference rule supplemented by the semantic labels. Let us consider it:

```
{P} prog {(label for_col substitution(Q, M,
                    (label rep_col rep(M, col(i, e1, e2), e(s, t))))))}
|- {any_predicate(P)} any_code(prog) for_col(M, i, k, e1, e2, e(s, t))
{any_predicate(Q)}
```

Note that using special inference rules for iteration over the matrix makes proving VCs easier than using more common inference rules in this case. It is easier to define the axiomatization of the replacement operation for iteration over the matrix because of an enormous amount of knowledge about matrices. Thus, this axiomatization has already been defined [15]. Given the verification of many linear algebra programs, it allows us to simplify their proving.

Noteworthy, using special inference rules for iteration over a matrix makes proving VCs easier than using the more common axiomatic semantics of the C-kernel programming language. Firstly, it is necessary to translate iteration over the matrix from *for* loop to *while* loop. Secondly, at least two additional VCs result from applying the inference rule for the *while* loop from the axiomatic semantics of the C-kernel language. Finally, it is necessary to “manually” define loop invariants to apply this semantics. The MetaVCG approach allows generating highly specialized versions of the VCG that will be based on special inference rules for linear algebra programs.

## 5 The Experiment

A program verification experiment has been performed to demonstrate the possibility of loop invariant elimination in practice. This experiment is based on the symbolic method of verification of definite iterations. The linear algebra program [14] that computes dot-product is considered in this experiment. The annotated function `dot_product` is the implementation of the function from the BLAS interface. This interface provides standard linear algebra procedures. A large number of linear algebra programs [16] are based on using this interface. Thus, the verification of such function implementation is important for programs that use it. Let us consider the C-light function `dot_product`:

```
/*@ requires (x != NULL) && (y != NULL) && (length > 0);
    ensures (value = DP_RESULT(0, length, x, y)); */
int dot_product(unsigned int length, int* x, int* y){
    int value = 0;
    for (unsigned int i = 0; i < length; i++){
        value += x[i] * y[i];}
    return value;}
```

This function calculates the dot-product of the vectors `x` and `y`. This dot-product accumulates in `value` variable. Thus, vector  $v$  from the symbolic method of verification of definite iterations consists of `value` variable in this case.

The language in which its specifications are defined is ACSL [6]. These specifications are present in usual C-commentaries to save program semantics.

The `requires` section in the ACSL language defines precondition and the `ensures` section in the ACSL language defines postcondition. The postcondition of this function is based on the `DP_RESULT` function. Let us consider the `DP_RESULT` function:

```
(declare-fun DP_RESULT (Int Int (Array Int Int) (Array Int Int)) Int)
(assert (and
  (forall ((i Int) (j Int) (x (Array Int Int)) (y (Array Int Int)))
    (implies (<= j i) (= (DP_RESULT i j x y) 0)))
  (forall ((i Int) (j Int) (x (Array Int Int)) (y (Array Int Int)))
    (implies (< i j) (= (DP_RESULT i j x y) (+
      (DP_RESULT i (- j 1) x y) (* (select x (- j 1)) (select y (- j 1))))))))))
```

The language defining this function is that of Z3, an automated theorem prover. This function provides a recursive definition of the dot-product of two vectors from the  $i$ -th to  $j$ -th element. This definition is based on using prefix notation.

The translator from C-light to C-kernel does not modify this program, because the translator recognizes the definite iteration. The inference rule for definite iteration [10] has been implemented in the metagenerator. Thus, the metagenerator yields the following VC for Z3. For the sake of simplicity, let us consider this VC without labels:

```
(assert (not (forall ((value Int) (length Int)
  (x (Array Int Int)) (y (Array Int Int)))
  (implies (> length 0) (= (rep length 0 x y)
    (DP_RESULT 0 length x y))))))
```

This VC is the negation of the statement that ensures the partial correctness of the Hoare triple. This statement is based on using `assert` construct. It allows us to define constraints on model. The negation is a feature of using SMT solvers such as Z3 [8].

This VC is based on the `rep` function. Therefore, the generator produces the recursive definition of the `rep` function for this program. Let us consider the `rep` function:

```
(declare-fun rep (Int Int (Array Int Int) (Array Int Int)) Int)
(assert (and
  (forall ((i Int) (value Int) (x (Array Int Int)) (y (Array Int Int)))
    (implies (<= i 0) (= (rep i value x y) value)))
  (forall ((i Int) (value Int) (x (Array Int Int)) (y (Array Int Int)))
    (implies (< 0 i) (= (rep i value x y) (+
      (rep (- i 1) value x y) (* (select x (- i 1)) (select y (- i 1))))))))))
```

It provides a recursive definition of the replacement operation that represents the loop effect in a symbolic form [10]. It allows us to avoid defining an invariant in this case. This definition is based on `implies` construct. It allows generator to define implication.

Z3 does not support induction [8]. Induction allows proving the theory that is based on the recursive definition of a function. Consequently, it is necessary

to use any constant value of length to prove this VC using Z3. The proof that is based on the statement that length is equal to 19 results in “unsat”. This result means that there is a correspondence between the program and its specifications in the case of length being equal to 19.

However, there are theorem provers that allow us to use induction. PVS provides one of such prover [17]. Moreover, PVS is the complex environment that contains the specification language, the predefined theories, the type checker, the interactive theorem prover and etc.

This language is based on typed higher-order logic. It allows assumptions, definitions, axioms and theorems to be organized into theories. Therefore, the translation from intermediate structures to this language has been implemented in the C-light project. Let us consider the theory for our example:

```
dot_product: THEORY
BEGIN

  DP_RESULT(i:nat, j:nat, x:ARRAY[nat->int],
            y:ARRAY[nat->int]): RECURSIVE int =
    IF j <= i THEN 0
    ELSE x(j-1) * y(j-1) + DP_RESULT(i, j-1, x, y) ENDIF
  MEASURE j

  rep(i:nat, value:int, x:ARRAY[nat->int],
      y:ARRAY[nat->int]): RECURSIVE int =
    IF i <= 0 THEN value
    ELSE x(i-1)*y(i-1)+rep(i-1, value, x, y) ENDIF
  MEASURE i

  vc: LEMMA
    FORALL (value:int, length:nat,
            x:ARRAY[nat -> int], y:ARRAY[nat -> int]):
      (0 < length) IMPLIES
      rep(length, 0, x, y) = DP_RESULT(0, length, x, y)

END dot_product
```

This theory contains the definition of DP\_RESULT function, the definition of rep function and the verification condition. The definition of VC is based on using implication. Note that it is necessary to define measure in the case of recursive function. This measure is an expression that decreases across recursive calls and is non-negative. It allows PVS system to avoid infinity recursion. The j parameter is the measure in the case of DP\_RESULT function and the i parameter is the measure in the case of rep function. The verification condition corresponds to lemma in this theory. Note that PVS is the interactive theorem prover [17]. Therefore, PVS tries to prove this lemma using strategy proposed by user.

This strategy consists of applying the inference procedures. PVS provides a collection of powerful primitive inference procedures that contains propositional and quantifier rules, induction, rewriting, simplification using decision procedures for equality and linear arithmetic and etc. Using (induct-and-simplify



"length") proof rule is the strategy in the case of our experiment. This is the special proof rule to prove theory in the case of induction. The application of this rule to this VC results in the following statement:

```
By induction on length, and by repeatedly rewriting and simplifying,
Q.E.D.
Run time = 0.14 secs.
Real time = 19.32 secs.
```

This result means that this theory has been proved. Consequently, there is a correspondence between the program and its specifications.

The (induct-and-simplify "length") rule reduces proof of initial formula to treatment of two cases: the induction basis and the induction step. Using (induct "length") rule instead of (induct-and-simplify "length") one may demonstrate these intermediate stages of proving. PVS automatically proves the induction basis and the induction step in the case of this experiment.

The almost automatic verification of dot\_product function is the important result at the current stage of work. Consequently, it has been decided to use PVS instead of Z3 in the C-light project.

This program is error-free. However, it is important to check the performance of the system of explanations of unproven VCs. The semantic mark-up method [3] allows the following explanation of this VC to be generated automatically:

```
This VC corresponds to lines 6-10 in the "dot_product" function.
Its purpose is to correctly represent the effect of definite iteration
over the arrays x, y.
Hence, given
the assumption that the precondition holds at line 6,
it is demonstrated that the value correctly represents the effect of
definite iteration over the arrays x, y at lines 7-10.
```

The generation of this explanation is based on semantic labels. Certain names and numbers are automatically inserted in this text by the metagenerator. The correspondence between this VC and the code is demonstrated by them. This explanation allows us to understand the aim and structure of this correctness condition.

This experiment allows us to check the performance of the C-light system. The VCs that are based on using the induction result from the application of symbolic method of verification of definite iterations. Thus, the experience of proving induction-based VC is important for us. The goal of this experiment has been achieved. Thus, this experiment is of great interest for us.

## 6 Conclusion

The primary purpose of the C-light project is to make verification practical in industrial settings. Automatic program error localization and explanation is very important in this context. (Otherwise, the user has to analyze "manually" unproved VCs.) The approach proposed by Denney and Fisher [3] has been

chosen to tackle this problem. The MetaVCG approach allows us to simplify the implementation of the semantic mark-up method in our project. The idea of this approach is based on a special extension of the Hoare rules by labels so that the axiomatic semantics itself can be used to generate explanations for unproved VCs.

Let us list some related research.

Firstly, the Centaur project [5] allows analyzing VCs using loop- and if-conditions. It is based on some algorithms of testing and debugging. Note that basic language of this system is very simple.

The mixed axiomatic semantics allows the symbolic method of verification of definite iterations to be implemented. This approach is under development by Maryasov and Nepomniaschy [8], it allows us to use a conventional VCG (but not a metagenerator). This semantics consists of the C-kernel programming language axiomatic semantics and some auxiliary inference rules. These rules allow generating simple VCs in some special cases. The VCG should identify these cases to apply these rules. These cases include definite iteration (3).

Frama-C tool is the part of WHY [4] project. This tool is environment for static analysis of C code. Jessie is a plugin for this tool. It is a deductive verification system. Note that it provides a common intermediate language for C and Java and translator from annotated C code to this intermediate language.

Boogie tool translates annotated programs to logical formulas in the VCC (A Verifier for Concurrent C) [2] project. Boogie provides Boogie PL intermediate language and VCG. Z3 is used to prove VCs. (Note that Boogie PL is not subset of the C language.) The correctness of the translation has not been proven in the case of VCC project and in the case of Jessie project.

Finally, Leino et al. [7] proposed to translate input language into a lower-level form for debugging. This form is extended by special labels from the underlying logic. Then explanations are generated for traces to safety conditions.

Let us consider topics for further research.

The application of the C-light system for verification of certain classes of programs, such as programs of engineering mathematics, is the most important task for future work. The MetaVCG approach [9] may be applied to any appropriate set inference rules and axioms extended by semantic labels. We are most interested in verification of linear algebra programs [14]. The symbolic method for verification of for-loops [10] should be applicable to such programs.

We plan also more verification case studies with search and sorting programs in general and counting sort in particular. The symbolic method of verifying definite iterations should allow us to avoid invariants in this case. In future we plan to try C-light system for verifying telecommunication protocols. It is worth noting that implementations of some protocols are based on using loops. Consequently, question about using the symbolic method of verifying definite iterations for simplifying verification for such programs is in the scope of our research interests.

## References

1. Apt, K.R., de Boer, F.S., Olderog, E.R.: Verification of sequential and concurrent programs, p. 450. Springer, London (1991). <https://doi.org/10.1007/978-1-84882-745-5>
2. Cohen, E., et al.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLS 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_2](https://doi.org/10.1007/978-3-642-03359-9_2)
3. Denney, E., Fischer, B.: Explaining verification conditions. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 145–159. Springer, Heidelberg (2008). [https://doi.org/10.1007/978-3-540-79980-1\\_12](https://doi.org/10.1007/978-3-540-79980-1_12)
4. Filliâtre, J.-C., Marché, C.: Multi-prover verification of C programs. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 15–29. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30482-1\\_10](https://doi.org/10.1007/978-3-540-30482-1_10)
5. Fraer, R.: Tracing the origins of verification conditions. In: Wirsing, M., Nivat, M. (eds.) AMAST 1996. LNCS, vol. 1101, pp. 241–255. Springer, Heidelberg (1996). <https://doi.org/10.1007/BFb0014320>
6. Kondratyev D.A.: The extension of the MetaVCG approach by semantic mark-up concept. In: Proceedings of the International Workshop Conferences on Tools & Methods of Program Analysis, St. Petersburg, pp. 107–118 (2015). (in Russian)
7. Leino, K.R.M., Millstein, T., Saxe, J.B.: Generating error traces from verification condition counterexamples. *Sci. Comput. Program.* **55**(1–3), 209–226 (2005)
8. Maryasov, I.V., Nepomniaschy, V.A.: Loop invariants elimination for definite iterations over unchangeable data structures in C programs. *Model. Anal. Inf. Syst.* **22**(6), 773–782 (2015)
9. Moriconi, M., Schwartz, R.L.: Automatic construction of verification condition generators from Hoare logics. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 363–377. Springer, Heidelberg (1981). [https://doi.org/10.1007/3-540-10843-2\\_30](https://doi.org/10.1007/3-540-10843-2_30)
10. Nepomniaschy, V.A.: Symbolic method of verification of definite iterations over altered data structures. *Program. Comput. Softw.* **31**(1), 1–9 (2005)
11. Nepomniaschy, V.A., Anureev, I.S., Promsky, A.V.: Towards verification of C programs: axiomatic semantics of the C-kernel languages. *Program. Comput. Softw.* **29**(6), 338–350 (2003)
12. Nepomniaschy, V.A., Anureev, I.S., Mikhailov, I.N., Promsky, A.V.: Towards verification of C programs. C-light language and its formal semantics. *Program. Comput. Softw.* **28**(6), 314–323 (2002)
13. Nepomniaschy, V.A., Anureev, I.S., Mikhailov, I.N., Promsky, A.V.: Verification-oriented language C-light. In: *System Informatics: Scientific Transactions/RAS. Siberian branch. Institute of Informatics Systems*, vol. 9, pp. 51–134 (2004)
14. Nepomniaschy, V.A., Ryakin, O.M.: Applied methods of program verification. In: *Radio and Communication*, 256 p. Moscow (1988). (in Russian)
15. Nepomniaschy, V.A., Sulimov, A.A.: Problem-oriented verification system and its application to linear algebra programs. *Theor. Comput. Sci.* **119**(1), 173–185 (1993)
16. Nepomniaschy, V.A., Sulimov, A.A.: Verification of the linear algebra programs in the system SPECTRUM. *Cybern. Syst. Anal.* **28**(5), 766–774 (1992)
17. Owre, S., Rushby, J.M., Shankar, N.: PVS: a prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992). [https://doi.org/10.1007/3-540-55602-8\\_217](https://doi.org/10.1007/3-540-55602-8_217)

# Highlights of the Rice-Shapiro Theorem in Computable Topology

Margarita Korovina<sup>1</sup>(✉) and Oleg Kudinov<sup>2</sup>

<sup>1</sup> A.P. Ershov Institute of Informatics Systems, SBRAS,  
Novosibirsk State University, Novosibirsk, Russia  
rita.korovina@gmail.com

<sup>2</sup> Sobolev Institute of Mathematics, SBRAS, Novosibirsk State University,  
Novosibirsk, Russia  
kud@math.nsc.ru

**Abstract.** Computable topological spaces naturally arise in computer science for continuous data type representations that have tools for effective reasoning about quite complex objects such as real numbers and functions, solutions of differential equations, functionals and operators. Algebraic and continuous domains, computable metric spaces, computable Polish spaces have been successfully used in the theoretical foundation of computer science. In this paper we consider generalisations of the famous Rice-Shapiro theorem in the framework of effectively enumerable topological spaces that contain the weakly-effective  $\omega$ -continuous domains and computable metric spaces as proper subclasses. We start with the classical case when the spaces admit principal computable numberings of computable elements and one can investigate arithmetical complexity of index sets. We provide requirements on effectively enumerable topological spaces which guarantee that the Rice-Shapiro theorem holds for the computable elements of these spaces. It turns out that if we relax these requirements then the Rice-Shapiro theorem does not hold. Then we discuss the perspective of extensions of the Rice-Shapiro theorem to spaces that do not have computable numberings of computable elements, in particular to computable Polish spaces.

**Keywords:** Continuous data type · Program semantics  
Arithmetical complexity · The Rice-Shapiro theorem

## 1 Introduction

One of the natural ideas in computable topology is an adaptation of the results from classical computability (recursion) theory concerning the lattice of computably enumerable sets to computable analysis for studying the structures of

---

The research leading to these results has received funding from the DFG grant WERA MU 1801/5-1 and the DFG/RFBR grant CAVER BE 1267/14-1 and 14-01-91334 and the grant council (under RF President) for State Aid of Leading Scientific Schools (grant NSh-6848.2016.1).

computable elements and effectively open sets of topological spaces. The major obstacle for the achievement of this goal is the different nature of discrete and continuous data. To overcome difficulties an approach is to figure out which particular classes of topological spaces fit better for generalisations of classical results. In this paper we propose to consider the theory of index sets as a promising candidate for merging classical recursion theory and computable topology. There are several reasons for doing this. One of them is that the theory of index sets provides methods for encoding problems in an effective way by natural numbers, i.e., generate the corresponding index sets which can be used for analysing their complexity in the settings of the Kleene-Mostowski arithmetical hierarchy. Another reason is that the theory of index sets has been already successfully employed in many areas in mathematics and computer science. In recursion theory index sets have been applied to get both new results and new elegant proofs of classical theorems such as the Post's theorem and the density theorem [11, 27, 29]. In computable model theory recent advancements are closely related to index sets [5, 6]. Following this direction in this paper we consider the famous Rice-Shapiro theorem. In classical computability theory the theorem gives a characterisation of subsets of natural numbers with computably enumerable index sets that provides a simple description of effectively enumerable properties of program languages in computer science.

Historically generalisations of the Rice-Shapiro theorem has been first proven for the algebraic domains [10], for the weakly effective  $\omega$ -continuous domains [33] and later on for the effectively pointed topological spaces that expand the weakly effective  $\omega$ -continuous domains [31].

We start with the case when the spaces admit principal computable numberings of computable elements and one can investigate arithmetical complexity of index sets. We are interested in principal computable numberings since they adequately reflect arithmetical complexity of subclasses of the computable elements, i.e., the index sets of a subclass with respect to any other computable numberings are  $m$ -reducible to the index set of the subclass with respect to the principal computable numbering. For example, if the index set of a subclass is computably enumerable (c.e.) with respect to the principal computable numbering then it is computably enumerable with respect to every computable numbering.

In this paper as promising candidates we consider effectively enumerable  $T_0$ -spaces with conditions on the family of basic neighborhoods of computable elements that guarantee the existence of a principal computable numbering. In [21] we already have shown that for this class of topological spaces generalisations of Rice's theorem hold. However these conditions are not sufficient to establish the Rice-Shapiro theorem. Towards the Rice-Shapiro theorem we enhance requirements on effectively enumerable  $T_0$ -spaces that result in a new class we call the class of modular  $T_0$ -spaces. It turns out that, on one hand, the Rice-Shapiro theorem holds for this class, on the other hand, this class extends the weakly effective  $\omega$ -continuous domains.

Finally, we explore possible extensions of the Rice-Shapiro theorem to spaces that do not have computable numberings of computable elements, in particular to the class of computable Polish spaces (*CPS*) that contains the real numbers, the Cantor space, the Baire space and many others spaces widely used in the foundation of computer science. Until now there have been no Rice-Shapiro-type theorems known that hold on the whole class *CPS*. There are two natural approaches how to avoid difficulties arising due the lack of computable numbering of computable elements. One approach is to embed the space under consideration to a wider space that admits a principal computable numbering of the computable elements and at the same time induces some properties of index sets to the original space. As examples of such spaces for embeddings we can consider the set of partial recursive functions for total recursive functions [27] in classical computability theory and the interval domain for the real numbers [1] in domain theory. Following this approach we embed the computable elements of a computable Polish space to the set of effective elements. We show that for any computable Polish space the set of effective elements admits principal computable numbering and the Rice-Shapiro theorem holds on them. As a corollary we get a weak Rice-Shapiro theorem for the computable elements. The other approach is to relax uniformity of the Rice-Shapiro theorem, i.e., consider subsets of computable elements that have computably enumerable index sets with respect to all computable sequences. We show that for the Baire space this approach does not work.

The paper is organised as follows. In Sect. 2 we give basic notions and definitions. We recall the notion of a computable element and conditions on the family of basic neighborhoods of computable elements that guarantee the existence of a principal computable numbering. Section 3 contains the definition of the class of modular  $T_0$ -spaces and the Rice-Shapiro theorem for this class. In Sect. 4 we explore possible extensions of the Rice-Shapiro theorem to spaces that do not have computable numberings of computable elements, in particular to the class of computable Polish spaces that contains the real numbers, the Cantor space, the Baire space and many others spaces widely used in the foundation of computer science.

## 2 Preliminaries

### 2.1 Recursion Theory

We refer the reader to [27, 28] for basic definitions and fundamental concepts of recursion theory and to [11] for basic definitions and fundamental concepts of numbering theory. We recall that, in particular,  $\varphi_e$  denotes the partial computable (recursive) function with an index  $e$  in the Kleene numbering. Let  $\varphi_e^s(x) = \varphi_e(x)$  if computation requires not more than  $s$  steps and  $x \leq s$ , otherwise  $\varphi_e^s(x) = -1$ . In this paper we also use notations  $W_e = \text{dom}(\varphi_e)$ ,  $\pi_e = \text{im}(\varphi_e)$  and  $W_e^s = \{x \mid \varphi_e^s(x) \geq 0\}$ . A sequence  $\{V_i\}_{i \in \omega}$  of computably enumerable sets is *computable* if  $\{(n, i) \mid n \in V_i\}$  is computably enumerable. It is worth noting

that this is equivalent to existence of a computable function  $f : \omega \rightarrow \omega$  such that  $V_i = W_{f(i)}$ . A numbering of a set  $Y$  is a total surjective map  $\gamma : \omega \rightarrow Y$ .

### 2.2 Weakly Effective $\omega$ -continuous Domains

In this section we present a background on domain theory. The reader can find more details in [14]. Let  $(D; \perp, \leq)$  be a partial order with a least element  $\perp$ . A subset  $A \subseteq D$  is *directed* if  $A \neq \emptyset$  and  $(\forall x, y \in A)(\exists z \in A)(x \leq z \wedge y \leq z)$ . We say that  $D$  is a *directed complete partial order*, denoted *depo*, if any directed set  $A \subseteq D$  has a supremum in  $D$ , denoted  $\bigsqcup A$ . For two elements  $x, y \in D$  we say  $x$  is *way-below*  $y$ , denoted  $x \ll y$ , if whenever  $y \leq \bigsqcup A$  for a directed set  $A$ , then there exists  $a \in A$  such that  $x \leq a$ . We say that  $\mathbf{B} \subseteq D$  is a *basis* (base) for  $D$  if for every  $x \in D$  the set  $\text{approx}_{\mathbf{B}}(x) = \{y \in \mathbf{B} \mid y \ll x\}$  is directed and  $x = \bigsqcup \text{approx}_{\mathbf{B}}(x)$ . We say that  $D$  is *continuous* if it has a basis; it is  $\omega$ -*continuous* if it has a countable basis.

**Definition 1.** [14] *Let  $\mathcal{D} = (D; \mathbf{B}, \beta, \leq, \perp)$  be an  $\omega$ -continuous domain where  $B$  is a basis,  $\beta : \omega \rightarrow B$  is a numbering of the basis. We say that  $\mathcal{D}$  is a weakly effective if the relation  $\beta(i) \ll \beta(j)$  is computably enumerable.*

### 2.3 Effectively Enumerable $T_0$ -spaces

In the paper we work with the effectively enumerable  $T_0$ -spaces. The class of effectively enumerable topological spaces has been proposed in [23]. This is a wide class containing the weakly effective  $\omega$ -continuous domains, computable metric spaces and positive predicate structures [22] that retains certain natural effectivity requirements which allow to represent important concepts of computability.

Let  $(X, \tau, \alpha)$  be a topological space, where  $X$  is a non-empty set,  $B \subseteq 2^X$  is a base of the topology  $\tau$  and  $\alpha : \omega \rightarrow B$  is a numbering.

**Definition 2.** [23] *A topological space  $(X, \tau, \alpha)$  is effectively enumerable if the following conditions hold.*

(1) *There exists a computable function  $g : \omega \times \omega \times \omega \rightarrow \omega$  such that*

$$\alpha(i) \cap \alpha(j) = \bigcup_{n \in \omega} \alpha(g(i, j, n)).$$

(2) *The set  $\{i \mid \alpha(i) \neq \emptyset\}$  is computably enumerable.*

In [23] it has been shown that the class of effectively enumerable topological spaces is a proper extension of the weakly effective  $\omega$ -continuous domains and the computable metric spaces. Further on we will often abbreviate  $(X, \tau, \alpha)$  by  $X$  if  $\tau$  and  $\alpha$  is clear from a context. We use the following notion of an effectively open set.

**Definition 3.** *x[23] Let  $(X, \tau, \alpha)$  be an effectively enumerable topological space.*

1. *A set  $\mathcal{O} \subseteq X$  is effectively open if there exists a computably enumerable set  $V$  such that  $\mathcal{O} = \bigcup_{n \in V} \alpha(n)$ .*
2. *A sequence  $\{\mathcal{O}_n\}_{n \in \omega}$  of effectively open sets is called computable if there exists a computable sequence  $\{V_n\}_{n \in \omega}$  of computably enumerable sets such that  $\mathcal{O}_n = \bigcup_{k \in V_n} \alpha(k)$ .*

Let  $\mathcal{O}_X$  denote the set of all open subsets of  $X$  and  $\mathcal{O}_X^e$  denote the set of all effectively open subsets of  $X$ .

**Definition 4.** *[20]*

1. *A numbering  $\beta : \omega \rightarrow \mathcal{O}_X^e$  is called computable if  $\{\beta(n)\}_{n \in \omega}$  is a computable sequence.*
2. *A numbering  $\beta : \omega \rightarrow \mathcal{O}_X^e$  is called principal computable if it is computable and every computable numbering  $\xi$  is computably reducible to  $\beta$ , i.e., there exists a computable function  $f : \omega \rightarrow \omega$  such that  $\xi(i) = \beta(f(i))$ .*

**Proposition 1.** *[20] For every effectively enumerable topological space there exists a principal computable numbering  $\alpha_X^e$  of  $\mathcal{O}_X^e$ .*

## 2.4 Computable Elements

In this section we work with effectively enumerable  $T_0$ -spaces  $(X, \tau, \alpha)$  and use the following notion of a computable element.

**Definition 5.** *[21] Let  $(X, \tau, \alpha)$  be an effectively enumerable  $T_0$ -space*

1. *An element  $x \in X$  is called computable if the set  $A_x = \{n \mid x \in \alpha(n)\}$  is computably enumerable.*
2. *A sequence  $\{a_n\}_{n \in \omega}$  of computable elements is called computable if the sequence  $\{A_{a_n}\}_{n \in \omega}$  of computably enumerable sets is computable.*

Let  $X_c$  be the set of all computable elements of an effectively enumerable  $T_0$ -space  $X$  and  $S_X = \{A_a \mid a \in X_c\}$ . It is easy to see that the definition above agrees with the notions of computable real number, computable element of a computable metric space [3], computable element of a weakly effective  $\omega$ -continuous domain [33, 36] and computable element of a computable topological space [16].

It is worth noting that there are effectively enumerable topological spaces without computable elements. As example we can consider the real numbers without the computable points. It is an effectively enumerable  $T_0$ -space, however there are no computable elements in this space.

**Definition 6.**

1. *A numbering  $\gamma : \omega \rightarrow X_c$  is called computable if  $\{A_{\gamma(n)}\}_{n \in \omega}$  is a computable sequence.*



2. A numbering  $\gamma : \omega \rightarrow X_c$  is called *principal computable* if it is computable and every computable numbering  $\xi$  is computably reducible to  $\alpha$ , i.e., there exists a computable function  $f : \omega \rightarrow \omega$  such that  $\xi(i) = \gamma(f(i))$ .

Now we address the natural question whether for an effectively enumerable space there exists a computable numbering of the computable elements. First, we observe that while for the computable real numbers there is no computable numbering [7, 24] as well as for the computable points of a complete computable metric space [3], for a weakly effective  $\omega$ -continuous domain there is a computable numbering of the computable elements [36]. Below we point out a natural sufficient condition on the family of basic neighborhoods of computable elements that guarantees the existence of a principal computable numbering. We recall that weakly effective  $\omega$ -continuous domains satisfy this condition.

**Definition 7.** [11] *Let  $S$  be a set of computably enumerable subsets of  $\omega$ . The set  $S$  is called a *wn-family* if there exists a partial computable function  $\sigma : \omega \rightarrow \omega$  such that*

- (i) if  $\sigma(n) \downarrow$  then  $W_{\sigma(n)} \in S$  and
- (ii) if  $W_n \in S$  then  $n \in \text{dom}(\sigma)$  and  $W_n = W_{\sigma(n)}$ .

From [11] it follows that any *wn-family*  $S_X$  has a standard principal computable numbering  $\gamma : n \mapsto W_{\sigma(h_0(n))}$ , where  $h_0 : \omega \rightarrow \omega$  is a total computable function such that  $\text{im}(h_0) = \text{dom}(\sigma)$ .

**Proposition 2.** [18] *Let  $(D; \mathbf{B}, \beta, \leq, \perp)$  be a weakly effective  $\omega$ -continuous domain. Then, the set  $S_D = \{ \{n \mid \beta(n) \ll a\} \mid a \in D_c \}$  is a *wn-family*.*

**Theorem 1.** [19] *Let  $(X, \tau, \alpha)$  be an effectively enumerable  $T_0$ -space and  $S_X = \{A_a \mid a \in X_c\}$ . If  $S_X$  is a *wn-family* then there exists an algorithm to construct a principal computable (canonical) numbering  $\bar{\gamma} : \omega \rightarrow X_c$ .*

*Proof.* Let us define  $\bar{\gamma}(n) = a \leftrightarrow A_a = \gamma(n)$ .

**Definition 8.** *Let  $\bar{\gamma} : \omega \rightarrow X_c$  be a principal computable numbering and  $L \subseteq X_c$ . The set  $Ix(L) = \{n \mid \bar{\gamma}(n) \in L\}$  is called an *index set* for the subset  $L$ .*

**Proposition 3.** [19] *Let  $(X, \tau, \alpha)$  be an effectively enumerable  $T_0$ -space and  $S_X$  be a *wn-family*. If  $K$  is effectively open in  $X_c$  then  $Ix(K)$  is computably enumerable.*

In [21] we already have shown that for an effectively enumerable topological space  $X$  such that  $S_X$  is a *wn-family* generalisations of Rice’s theorem hold. Moreover from the results in [11] it is easy to see the following. If  $X_c$  has the least element then the principal computable numbering is complete. So several results from classical numbering theory can be generalised for this case, in particular, if  $Ix(K)$  is computably enumerable then it is creative. Hence at the first glance this class looks promising to generalise the Rice-Shapiro theorem. However there is a counterexample that shows the existence of an effectively enumerable topological

space  $X$  such that  $S_X$  is a  $wn$ -family but the Rice-Shapiro theorem does not hold [18]. This forces us to search for stronger requirements on effectively enumerable topological spaces which do not restrict the class too much and at the same time guarantee that the Rice-Shapiro theorem holds.

### 3 The Rice-Shapiro Theorem for Modular $T_0$ -spaces

In this section we recall the generalised Rice-Shapiro Theorem for the class of modular  $T_0$ -spaces. Below We use the specialisation order

$$x \leq y \iff \text{for all open } \mathcal{O} \text{ if } x \in \mathcal{O} \text{ then } y \in \mathcal{O}.$$

For  $B \subseteq X$  we use notation  $x \leq B$  if, for all  $y \in B, x \leq y$ .

**Definition 9.** [18] *An effectively enumerable  $T_0$ -space  $(X, \tau, \alpha)$  is called a modular  $T_0$ -space if it satisfies the following requirements:*

*Req 1:  $S_X$  is a  $wn$ -family.*

*Req 2: There exist a computable sequence  $\{b_n\}_{n \in \omega}$  of computable elements and a computable sequence  $\{\mathcal{O}_n\}_{n \in \omega}$  of effectively open sets such that*

- (a)  $b_n \leq \mathcal{O}_n$ , where  $\leq$  is the specialisation order, and*
- (b) for all  $m \in \omega \alpha(m) = \bigcup_{b_i \in \alpha(m)} \mathcal{O}_i$ .*

**Proposition 4.** [18] *Any weakly effective  $\omega$ -continuous domain is a modular  $T_0$ -space.*

**Theorem 2 (Generalised Rice-Shapiro).** [18] *Let  $\mathcal{X}$  be a modular  $T_0$ -space and  $K \subseteq X_c$ . Then  $Ix(K)$  is computably enumerable if and only if  $K$  is effectively open in  $X_c$ .*

As a straightforward corollary we get the well-known result that for computable elements of a weakly effective  $\omega$ -continuous domain the Rice-Shapiro theorem holds.

### 4 The Rice-Shapiro Theorem for CPS

In this section we work with the following notion of a computable Polish space abbreviated as *CPS*. A computable Polish space is a complete separable metric space  $\mathcal{X}$  without isolated points and with a metric  $d : X \times X \rightarrow \mathbb{R}$  such that there is a countable dense subset  $\mathcal{B} = \{b_1, b_2, \dots\}$  called a *basis of  $\mathcal{X}$*  with the numbering  $\beta$  that makes the following two relations:  $\{(n, m, i) \mid d(b_n, b_m) < q_i, q_i \in \mathbb{Q}\}$  and  $\{(n, m, i) \mid d(b_n, b_m) > q_i, q_i \in \mathbb{Q}\}$  computably enumerable (c.f. [37]). The standard notations  $B(x, y)$  and  $\overline{B}(x, y)$  are used for open and closed balls with the center  $x$  and the radius  $y$ . We also use the notation  $\beta(n) = b_n$  for a numbering  $\beta : \omega \rightarrow \mathcal{B}$ .

For a computable Polish space  $(X, \mathcal{B}, d)$  in a natural way we define the numbering of the base of the standard topology as follows.

$$\begin{aligned} \alpha(0) &= \emptyset, \\ \alpha(i) &= B(b_{u(i)}, r_i) \text{ for } i > 0, \text{ where} \end{aligned}$$

$u : \omega \rightarrow \omega$  is a suitable computable function,  $\{r_i\}_{i \in \omega}$  is a computable sequence of all positive rational numbers such that  $\{(u(i), r_i) \mid i > 0\} = (\omega \setminus \{0\}) \times \mathbb{Q}^+$ . It is easy to see that  $(X, \tau, \alpha)$  is an effectively enumerable topological space. Therefore we consider the computable Polish spaces as a proper subclass of the effectively enumerable topological spaces. For details we refer to [23]. We use the Baire space  $\mathcal{N} = (\omega^\omega, \tau_{\mathcal{N}})$  defined as follows.

$$\begin{aligned} \omega^0 &= \{\perp\}, \text{ where } \perp \text{ is the empty word,} \\ \omega^{<\omega} &= \bigcup_{n \in \omega} \omega^n, \\ \omega^\omega &= \{f \mid f : \omega \rightarrow \omega\} \\ &\text{(informally, the set of all paths in the tree } \omega^{<\omega}\text{).} \end{aligned}$$

The standard topology  $\tau_{\mathcal{N}}$  on  $\omega^\omega$  is generated by the base that contains all clopen sets of the type

$$\mathfrak{A}_w = \{f \in \mathcal{N} \mid f[s] = w, s = \text{length}(w)\},$$

where  $w \in \omega^{<\omega}$  and the interpretation of  $f[s]$  is as follows:

$$\begin{aligned} f[0] &= \perp, \\ f[s] &= \langle f(0), \dots, f(s-1) \rangle. \end{aligned}$$

It is easy to verify that the topology on  $\mathcal{N}$  is generated by the metric

$$d(f, g) = \begin{cases} 0 & \text{if } f = g \\ \frac{1}{\text{least } n[f(n) \neq g(n)] + 1} & \text{if } f \neq g. \end{cases}$$

Below we use the following notations  $\mathbb{R}^+ \Leftrightarrow \{r \mid r \in \mathbb{R} \text{ and } r \geq 0\}$  and  $\mathbb{Q}^{>0} \Leftrightarrow \{q \mid q \in \mathbb{Q} \text{ and } q > 0\}$ .

### 4.1 A Weak version of the Rice-Shapiro theorem for CPS

In this section we embed the computable elements of a computable Polish space in the set of effective elements. On intermediate steps for a computable Polish space  $(M, d, \mathcal{B})$  we construct a lifted domain  $\mathcal{D}^{\mathcal{M}}$  that is a weakly effective  $\omega$ -continuous domain such that the Scott topology on  $\mathcal{D}^{\mathcal{M}}$  agrees with the metric  $d$  and the set of maximal elements is homeomorphic to the original space. Then in a natural way we map the maximal computable elements of the lifted domain to the effective elements of the original space. We show that for any computable Polish space the set of effective elements admits principal computable numbering and the Rice-Shapiro theorem holds on them. As a corollary we get a weak Rice-Shapiro theorem for the computable elements.

**Theorem 3.** For any  $\mathcal{M} = (M, d, \mathcal{B}) \in \text{CPS}$  one can effectively construct a weakly effective  $\omega$ -continuous domain  $\mathcal{D}^{\mathcal{M}} = (D^{\mathcal{M}}; \mathbf{B}, \gamma, \leq)$  called a lifted domain for  $\mathcal{M}$  such that

- $M$  is homeomorphic to the set of all maximal elements of  $D$  with respect to the specialisation order on  $D^{\mathcal{M}}$ .
- The standard topology on  $\mathcal{M}$  coincides with the topology induced by the Scott topology on  $D^{\mathcal{M}}$ .

We give an outline of the proof based on the following propositions and observations. First we define

$$D^{\mathcal{M}} \rightleftharpoons M \times \mathbb{R}^+ = \{(a, r) \mid a \in M \text{ and } r \in \mathbb{R}^+\};$$

$$(b, q) \leq (a, r) \rightleftharpoons d(a, b) + r \leq q;$$

$$\mathbf{B} \rightleftharpoons \{(a, q) \mid a \in \mathcal{B} \text{ and } q \in \mathbb{Q}^{>0}\};$$

$\gamma : \omega \rightarrow \mathbf{B}$  is the numbering induced by  $\beta$  and the standard numbering of  $\mathbb{Q}^{>0}$ .

*Remark 1.* On the real numbers  $\mathbb{R}$  with the standard metric there is a one-to-one correspondence between the pair  $(a, r)$  and the closed ball  $\overline{B}(a, r)$  and the relation  $\leq$  coincides with the inverse non-strict inclusion, i.e.,  $(b, q) \leq (a, r) \leftrightarrow \overline{B}(a, r) \supseteq \overline{B}(b, q)$ . Unfortunately, in general on a computable Polish space there is no such intuitive natural correspondence.

**Lemma 1.**  $(D^{\mathcal{M}}, \leq)$  is a dcpo.

*Proof.* We prove that any directed set has a least upper bound. Let  $\mathcal{A}$  be a directed subset of  $D_{\mathcal{M}}$ . For convenience we bijectively associate  $\mathcal{A}$  with  $(I, \leq_I)$ , where  $i \leq_I j \leftrightarrow (a_i, r_i) \leq (a_j, r_j)$ . Therefore we can write  $\mathcal{A} = \{(a_i, r_i)\}_{i \in I}$ . Since  $\{r_i\}_{i \in I}$  is also directed, put  $\inf(\{r_i \mid i \in I\}) = \epsilon \geq 0$ . From  $(a_i, r_i - \epsilon) \leq (a_j, r_j - \epsilon)$ , for  $i \leq_I j$ , it follows that  $\{a_i\}_{i \in I}$  is a Cauchy net therefore  $a_i \rightarrow a \in M$ . Let us prove that  $(a, \epsilon) = \sup(\{(a_i, r_i) \mid i \in I\})$ . By definition, for  $j \geq_I i$ ,  $d(a_j, a_i) + r_j \leq r_i$ . Passing to the limit on the index  $j$  we have  $d(a, a_i) + \epsilon \leq r_i$ . So,  $(a, \epsilon) \geq (a_i, r_i)$  for any  $i \in I$ . Now we assume that  $(b, R)$  is an upper bound of  $\{(a_i, r_i) \mid i \in I\}$ . So, for all  $i \in I$ ,  $d(b, a_i) + R \leq r_i$ . Passing to the limit on the index  $i$  we have  $d(b, a) + R \leq \epsilon$ , i.e.,  $(b, R) \geq (a, \epsilon)$ . Therefore  $(a, \epsilon)$  is the least upper bound.

**Lemma 2.** For the way-below relation  $\ll$  the following holds.

$$(b, q) \ll (a, r) \leftrightarrow d(a, b) + r < q.$$

The sub-basis of topology  $\tau_{D^{\mathcal{M}}}$  is the set of open sets in the form

$$U_{n,q} = \{(b, r) \mid (b, r) \gg (\beta(n), q), \text{ where } \beta(n) \in \mathcal{B} \text{ and } q \in \mathbb{Q}^{>0}\}.$$

**Lemma 3.** *The topology  $\tau_{D^{\mathcal{M}}}$  is the Scott topology on  $D^{\mathcal{M}}$ .*

Let  $D^{\mathcal{M},\perp}$  denote  $D^{\mathcal{M}} \cup \{\perp\}$ , where  $\perp \ll a$  for all  $a \in D^{\mathcal{M}}$ . We can chose  $\perp$  as  $\{M\}$ .

**Proposition 5.** *The space  $\mathcal{D}^{\mathcal{M},\perp} = (D^{\mathcal{M},\perp}; \mathbf{B}, \gamma, \leq)$  is a weakly effective  $\omega$ -continuous domain such that the set of maximal elements of  $D^{\mathcal{M},\perp}$  is homeomorphic to  $\mathcal{M}$  and  $S_{D^{\mathcal{M},\perp}}$  is a wn-family with respect to the Scott topology.*

*Proof.* The claim follows from Lemmas 1, 2 and 3.

**Corollary 1.** *There exists a principal computable numbering  $\nu : \omega \rightarrow D_c^{\mathcal{M},\perp}$  and the Rice-Shapiro theorem holds.*

*Proof.* The claim follows from the results in [33].

Now we introduce the notion of an effective element of  $\mathcal{M}$  and show that the set of effective elements  $M_{eff}$  admits a principal computable numbering. Let  $r : \omega \rightarrow \mathbb{Q}$  be the standard computable numbering of  $\mathbb{Q}$ .

**Definition 10.** *Let  $\mathcal{M} = (M, d, \mathcal{B}) \in CPS$ . An element  $x \in M$  is called  $\epsilon$ -effective if the set  $\{(n, m) \mid \beta(n) \in \mathcal{B}, r(m) \in \mathbb{Q}^{>0} \text{ and } d(x, \beta(n)) + \epsilon < r(m)\}$  is computably enumerable.*

**Definition 11.** *Let  $\mathcal{M} = (M, d, \mathcal{B}) \in CPS$ . An element  $x \in M$  is called effective if there exists a right-computable number  $\epsilon \geq 0$ , i.e.,  $\{n \mid r(n) > \epsilon\}$  is computably enumerable, such that  $x$  is  $\epsilon$ -effective.*

*Remark 2.* It is worth noting that the effective real numbers contain right-computable and left-computable reals. Moreover, the effective elements are exactly sums of right- and left-computable reals. At first glance it looks natural for embeddings of the reals to consider either the left-computable or the right-computable reals. However the main obstacle is that while admitting computable numberings the left-computable and the right-computable reals don't admit principal computable numberings.

- Definition 12.**
1. *A numbering  $\alpha : \omega \rightarrow M_{eff}$  is computable if, for some computable sequence  $\{\epsilon_n\}_{n \in \omega}$  of right-computable reals, the set  $\{(n, m, k) \mid \beta(m) \in \mathcal{B}, r(k) \in \mathbb{Q}^{>0} \text{ and } d(\alpha(n), \beta(m)) + \epsilon_n < r(k)\}$  is computably enumerable.*
  2. *A numbering  $\gamma : \omega \rightarrow M_{eff}$  is called principal computable if it is computable and every computable numbering  $\xi$  is computably reducible to  $\alpha$ , i.e., there exists a computable function  $f : \omega \rightarrow \omega$  such that  $\xi(i) = \gamma(f(i))$ .*

The principal computable numbering of  $M_{eff}$  can be constructed as the following composition:

$$\omega \xrightarrow{\nu} D_c^{\mathcal{M},\perp} \xrightarrow{\tau} M_{eff},$$

$\tau$  is a Borel function such that  $\tau((a, r)) = a$  and  $\tau(\perp) = \beta(0)$ .

It is worth noting that  $\tau$  maps any computable sequence of elements of  $D_c^{\mathcal{M},\perp}$  to a computable sequence of elements of  $M_{eff}$ .

**Proposition 6.** *Let  $\mathcal{M} = (M, d, \mathcal{B}) \in \text{CPS}$ . Then  $K \subseteq M_{\text{eff}}$  is effectively open in  $M_{\text{eff}}$  if and only if  $Ix(K)$  is computably enumerable.*

*Proof.* The claim follows from Theorem 2, Corollary 1 and the construction above.

**Theorem 4 (Weak Rice-Shapiro Theorem).** *Let  $\mathcal{M} = (M, d, \mathcal{B}) \in \text{CPS}$ . Then  $K \subseteq M_c$  is effectively open in  $M_c$  if and only if there exists  $\tilde{K} \subseteq M_{\text{eff}}$  such that  $Ix(\tilde{K})$  is computably enumerable and  $\tilde{K} \cap M_c = K$ .*

*Proof.* The claim follows from Proposition 6.

*Remark 3.* It is worth noting that for a recursive Polish space [17, 26] the index set of computable elements is  $\Pi_2^0$ -complete with respect to the principal computable numbering of the effective elements. However we conjecture that for some computable Polish spaces the set can be  $\Pi_3^0$ -complete.

## 4.2 Towards a Non-uniform Rice-Shapiro Theorem

We discuss the perspective of a non-uniform version of the Rice-Shapiro theorem for spaces that do not have computable numberings of computable elements, in particular to the class of computable Polish spaces that contains the real numbers, the Cantor space and the Baire space.

The following definition looks like a natural generalisation of the subsets of  $X_c$  with computably enumerable index sets when the space lacks a computable numbering of computable elements.

**Definition 13.** *Let  $\mathcal{X} = (X, \tau, \alpha)$  be an effectively enumerable  $T_0$ -space and  $X_c$  be the set of computable elements. Then  $K \subseteq X_c$  is called intrinsically computably enumerable if for all computable sequences  $\{x_n\}_{n \in \omega}$ , where  $x_n \in X_c$ , the set  $\{k \mid x_k \in K\}$  is computably enumerable.*

*Conjecture 1.* (Non-uniform Rice-Shapiro Theorem) The set  $K \subseteq X_c$  is effectively open in  $X_c$  if and only if  $K$  is intrinsically computably enumerable.

**Proposition 7.** *For any effectively enumerable  $T_0$ -space if  $K \subseteq X_c$  is effectively open in  $X_c$  then  $K$  is intrinsically computably enumerable.*

Now we consider the Baire space  $\mathcal{N} = (\omega^\omega, \tau_{\mathcal{N}})$  with the standard topology.

**Theorem 5.** *There exists an intrinsically computably enumerable set  $K \subseteq \mathcal{N}_c$  that is not effectively open in  $\mathcal{N}_c$ . Therefore Conjecture 1 does not hold on  $\mathcal{N}$ .*

The outline of our proof for the Baire space is based on the following propositions and observation. We consider the Sierpinski space  $\mathbb{S} = (\{0, 1\}, \tau_{\mathbb{S}})$  with the standard topology  $\tau_{\mathbb{S}} = \{\emptyset, \{1\}, \{0, 1\}\}$ . The key idea of the proof is based on the closed relations between Banach–Mazur–computable functions  $F : \mathcal{X} \rightarrow \mathbb{S}$  and intrinsically computably enumerable subsets  $K \subseteq X$ , between effectively

open in  $X_c$  subsets  $K \subseteq X$  and total computable functions  $G : \mathcal{X} \rightarrow \mathbb{S}$ . Therefore by constructing a Banach–Mazur–computable function  $F_K : \mathcal{N} \rightarrow \mathbb{S}$  with certain properties that does not have a total computable continuation on  $\mathcal{N}$  we will prove Theorem 5.

By analogy to [25] we give the notion of Banach-Mazur–computable function over effectively enumerable  $T_0$ -spaces.

**Definition 14.** *Let  $\mathcal{X}$  and  $\mathcal{Y}$  be effectively enumerable  $T_0$ -spaces. A function  $f : \mathcal{X} \rightarrow \mathcal{Y}$  is called Banach-Mazur–computable if  $\text{dom}(f) \supseteq X_c$  and it maps any computable sequence of computable elements of  $X$  to a computable sequence of computable elements of  $Y$ .*

For the notion of total computable functions over effectively enumerable  $T_0$ -spaces we refer to [19]. Below we use the following property.

**Proposition 8.** [19] *Let  $\mathcal{X} = (X, \tau, \alpha)$  be an effectively enumerable topological space and  $\mathcal{Y} = (Y, \lambda, \beta)$  be an effectively enumerable  $T_0$ -space. A total function  $f : \mathcal{X} \rightarrow \mathcal{Y}$  is computable if and only if  $f$  is effectively continuous.*

**Lemma 4.** *Let  $\mathcal{X} = (X, \tau, \alpha)$  be an effectively enumerable topological space,  $F : \mathcal{X} \rightarrow \mathbb{S}$  be a partial function such that  $\text{dom}(F) \subseteq X_c$  and  $K = \{x \in X_c \mid F(x) = 1\}$ . Then the following assertions hold.*

1. *The set  $K$  is intrinsically computably enumerable if and only if the function  $F$  is Banach-Mazur–computable.*
2. *The set  $K$  is effectively open in  $X_c$  if and only if the function  $F$  coincides on  $X_c$  with a total computable function  $G : \mathcal{X} \rightarrow \mathbb{S}$ .*

In order to construct a required function we take the well-known Friedberg function [12, 27]  $F : \mathcal{N} \rightarrow \omega$  such that  $\text{dom}(F) = \mathcal{N}_c$  and it is Banach-Mazur–computable but does not have a total computable continuation on  $\mathcal{N}$ , i.e., there is no a total computable function  $G : \mathcal{N} \rightarrow \mathbb{S}$  such that  $G \upharpoonright_{\mathcal{N}_c} = F \upharpoonright_{\mathcal{N}_c}$ . In fact, it means that  $\{F^{-1}(n)\}_{n \in \omega}$  is not a computable sequence of effectively open sets in  $\mathcal{N}_c$ . For our construction we use the computable function  $Delay : \mathcal{N} \rightarrow \mathcal{N}$  defined as  $Delay(f)(n) = f(n + 1)$ , i.e.,  $Delay$  removes  $f(0)$  and shifts the arguments of  $f$ . Let us define  $\tilde{F} : \mathcal{N} \rightarrow \mathbb{S}$  by the following rules:

$$\tilde{F}(f) = \begin{cases} 1 & \text{if } F(Delay(f)) = f(0) \\ \uparrow & \text{if } f \notin \mathcal{N}_c \\ 0 & \text{otherwise.} \end{cases}$$

It is easy to see that  $\tilde{F}$  is a Banach-Mazur–computable function, i.e., it maps a computable sequence of computable elements of  $\mathcal{N}$  to a computable sequence of computable elements of  $\mathbb{S}$ . However, as well as  $F$  the function  $\tilde{F}$  does not have a computable continuation on  $\mathcal{N}$ . Indeed, assume that such continuation  $G : \mathcal{N} \rightarrow \mathbb{S}$  exists. Then  $\tilde{G}^{-1}(1) \cap \mathcal{N} = \{Con(n, g) \mid g \in F^{-1}(n)\}$  should be effectively open in  $\mathcal{N}$ , where  $Con(n, g) = f$  and

$$f(x) = \begin{cases} n & \text{for } x = 0 \\ g(x - 1) & \text{for } x > 0. \end{cases}$$

Hence  $\{F^{-1}(n)\}_{n \in \omega}$  is a computable sequence of effectively open sets in  $\mathcal{N}_c$ . This contradicts the choice of  $F$ . So, by Lemma 4, the set  $K = \{x \in \mathcal{N}_c \mid \tilde{F}(x) = 1\}$  is intrinsically computably enumerable but not effectively open in  $\mathcal{N}_c$ .

For the real numbers  $\mathbb{R}$ , we hope that a proof can be done in a similar way using results from [13].

## 5 Conclusion

In this paper we showed the generalised Rice-Shapiro theorem for computable elements in the framework of modular  $T_0$ -spaces that are a wide class of effectively enumerable topological spaces which contains the weakly effective  $\omega$ -continuous domains as a proper subclass. For the computable Polish spaces we also gave a hint how to get a weak Rice-Shapiro theorem. We discussed the perspective of a non-uniform version of the Rice-Shapiro theorem. It will be challenging to give an answer to Conjecture 1 for the whole computable Polish spaces.

## References

1. Edalat, A.: Domains for computation in mathematics, physics and exact real arithmetic. *Bull. Symbolic Logic* **3**(4), 401–452 (1997)
2. Berger, U.: Total sets and objects in domain theory. *Ann. Pure Appl. Logic* **60**(2), 91–117 (1993)
3. Brattka, V.: Computable versions of Baire’s category theorem. In: Sgall, J., Pultr, A., Kolman, P. (eds.) *MFCS 2001*. LNCS, vol. 2136, pp. 224–235. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-44683-4\\_20](https://doi.org/10.1007/3-540-44683-4_20)
4. Brodhead, P., Cenzer, D.A.: Effectively closed sets and enumerations. *Arch. Math. Log.* **46**(7–8), 565–582 (2008)
5. Calvert, W., Fokina, E., Goncharov, S.S., Knight, J.F., Kudinov, O.V., Morozov, A.S., Puzarenko, V.: Index sets for classes of high rank structures. *J. Symb. Log.* **72**(4), 1418–1432 (2007)
6. Calvert, W., Harizanov, V.S., Knight, J.F., Miller, S.: Index sets of computable structures. *J. Algebra Logic* **45**(5), 306–325 (2006)
7. Ceitin, G.S.: Mean value theorems in constructive analysis. *Transl. Am. Math. Soc. Transl. Ser.* **2**(98), 11–40 (1971)
8. Cenzer, D.A., Remmel, J.B.: Index sets for  $\Pi_1^0$  classes. *Ann. Pure Appl. Logic* **93**(1–3), 3–61 (1998)
9. Cenzer, D.A., Remmel, J.B.: Index sets in computable analysis. *Theor. Comput. Sci.* **219**(1–2), 111–150 (1999)
10. Ershov, Y.L.: Model  $\mathbb{C}$  of partial continuous functionals. In: *Logic Colloquium 76*, pp. 455–467. North-Holland, Amsterdam (1977)
11. Ershov, Y.L.: Theory of numberings. In: Griffor, E.R. (ed.) *Handbook of Computability Theory*, pp. 473–503. Elsevier Science B.V., Amsterdam (1999)
12. Friedberg, R.M.: 4-quantifier completeness: a Banach-Mazur functional not uniformly partial recursive. *Bulletin de l’Academie Polonaise des sciences, Serie des sci. math., astr. et phys.* **6**(1), 1–5 (1958)



13. Hertling, P.: A Banach-Mazur computable but not Markov computable function on the computable real numbers. *Ann. Pure Appl. Logic* **132**(2–3), 227–246 (2005)
14. Gierz, G., Heinrich Hofmann, K., Keime, K., Lawson, J.D., Mislove, M.W.: Continuous lattices and domain. In: *Encyclopedia of Mathematics and its Applications*, vol. 93. Cambridge University Press, Cambridge (2003)
15. Grubba, T., Weihrauch, K.: On computable metrization. *Electr. Notes Theor. Comput. Sci.* **167**, 345–364 (2007)
16. Grubba, T., Weihrauch, K.: Elementary computable topology. *J. UCS* **15**(6), 1381–1422 (2009)
17. Gregoriades, V., Kispeter, T., Pauly, A.: A comparison of concepts from computable analysis and effective descriptive set theory. *Math. Struct. Comput. Sci.* (2014). <http://arxiv.org/abs/1403.7997>
18. Korovina, M., Kudinov, O.: Rice-Shapiro Theorem in Computable Topology (2017). <http://arxiv.org/abs/1708.09820>
19. Korovina, M., Kudinov, O.: Computable elements and functions in effectively enumerable topological spaces. *J. Math. Struct. Comput. Sci.* (2016). <https://doi.org/10.1017/S0960129516000141>
20. Korovina, M., Kudinov, O.: Index sets as a measure of continuous constraint complexity. In: Voronkov, A., Virbitskaite, I. (eds.) *PSI 2014. LNCS*, vol. 8974, pp. 201–215. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46823-4\\_17](https://doi.org/10.1007/978-3-662-46823-4_17)
21. Korovina, M., Kudinov, O.: Rice’s theorem in effectively enumerable topological spaces. In: Beckmann, A., Mitran, V., Soskova, M. (eds.) *CiE 2015. LNCS*, vol. 9136, pp. 226–235. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-20028-6\\_23](https://doi.org/10.1007/978-3-319-20028-6_23)
22. Korovina, M., Kudinov, O.: Positive predicate structures for continuous data. *J. Math. Struct. Comput. Sci.* **25**(8), 1669–1684 (2015)
23. Korovina, M., Kudinov, O.: Towards computability over effectively enumerable topological spaces. *Electr. Notes Theor. Comput. Sci.* **221**, 115–125 (2008)
24. Martin-Löf, P.: *Notes on Constructive Mathematics*. Almqvist & Wiksell, Stockholm (1970)
25. Mazur, S.: *Computable Analysis*, vol. 33. *Razprawy Matematyczne*, Warsaw (1963)
26. Moschovakis, Y.N.: Recursive metric spaces. *Fund. Math.* **55**, 215–238 (1964)
27. Rogers, H.: *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, New York (1967)
28. Soare, R.I.: *Recursively Enumerable Sets and Degrees: A Study of Computable Functions and Computably Generated Sets*. Springer, Heidelberg (1987)
29. Shoenfield, J.R.: *Degrees of Unsolvability*. North-Holland Publishing Company, Amsterdam (1971)
30. Spreen, D.: On effective topological spaces. *J. Symb. Log.* **63**(1), 185–221 (1998)
31. Spreen, D.: Effective inseparability in a topological setting. *Ann. Pure Appl. Logic* **80**(3), 257–275 (1996)
32. Spreen, D.: On some decision problems in programming. *Inf. Comput.* **122**(1), 120–139 (1995)
33. Spreen, D.: On r.e. inseparability of CPO index sets. In: Börger, E., Hasenjaeger, G., Rödding, D. (eds.) *LaM 1983. LNCS*, vol. 171, pp. 103–117. Springer, Heidelberg (1984). [https://doi.org/10.1007/3-540-13331-3\\_36](https://doi.org/10.1007/3-540-13331-3_36)
34. Vjugin, V.V.: On some examples of upper semilattices of computable numberings. *Algebra Logic* **13**(5), 512–529 (1973)
35. Weihrauch, K.: *Computable Analysis*. Springer, Heidelberg (2000)

36. Weihrauch, K., Deil, T.: Berechenbarkeit auf cpo-s. Schriften zur Angew. Math. u. Informatik **63**. RWTH Aachen (1980)
37. Weihrauch, K.: Computability on computable metric spaces. Theor. Comput. Sci. **113**(1), 191–210 (1993)

# A Memory Model for Deductively Verifying Linux Kernel Modules

Mikhail Mandrykin<sup>1</sup>(✉) and Alexey Khoroshilov<sup>1,2,3,4</sup>

<sup>1</sup> Institute for System Programming of the Russian Academy of Sciences,  
Moscow, Russia

{mandrykin,khoroshilov}@ispras.ru

<sup>2</sup> The Faculty of Computational Mathematics and Cybernetics of Lomonosov  
Moscow State University, Moscow, Russia

<sup>3</sup> Moscow Institute of Physics and Technology, Dolgoprudny, Russia

<sup>4</sup> National Research University Higher School of Economics, Moscow, Russia

**Abstract.** Several previous evaluations of memory models for SMT-based deductive verification tools have shown that the choice of memory model may significantly affect both the number of automatically discharged verification conditions and the capabilities of the verification tool. One of the most efficient memory models for deductive verification of low-level C code is based on region analysis and component-as-array modeling. However, originally this model doesn't support many C language idioms widely used in low-level system code including the Linux kernel. The paper suggests a modification of this model that extends it with full support for arbitrarily nested structures, unions and arrays, arbitrary pointer arithmetic and general pointer type casts. The extension for nested structures and arrays requires no additional annotation overhead. The support for pointer arithmetic, unions and pointer type casts generally requires user annotations. The proposed model fully preserves the performance of the original memory model for earlier supported code. Preliminary practical evaluation on an industrial security kernel module showed a small specification overhead required for code where the proposed model is not fully automatic.

**Keywords:** Deductive verification · Memory model · Region analysis

## 1 Introduction

Reasoning about memory separation is a well known problem in the field of imperative program analysis and verification. In particular, there are two major categories of approaches to dealing with memory separation in the context of deductive verification.

The overall idea of approaches in the first category basically consists in finding a sufficiently expressive and at least partially automatizable logical fragment relevant to the target code, then restricting the logic allowed in supported

---

The research was supported by RFBR grant 15-01-03934.

specifications to this fragment, and finally employing the developed automation techniques (decision procedures and/or type checkers). The corresponding approaches are successfully implemented in tools supporting various fragments of separation logic [1] e.g. Logic of Graph Reachability with Stratified Sets (GRASS) [2,3] or a fragment of separation logic with fractional permissions [4], and fragments of other logics for recursive data structures (often equivalent to some separation logic fragments) e.g. the Logic of Interpreted Sets and Bounded Quantification (LISBQ) [5]. The most notable advantage of these approaches is resulting concision of specifications and (possibly) proofs while the most important limitation is the expressiveness of the supported logics.

In scope of deductive verification for low-level system code the most important limitations of these approaches are related to handling of arrays, some specific uses of pointer arithmetic (e.g. obtaining the address of the outer structure by subtracting the corresponding offset from the address of a nested structure), pointer type casts (e.g. `(char *)p` when `p` has type `int *`) and casts between pointer types and the integer type of the corresponding size (e.g. using `(unsigned long)p` in order to save a pointer into a general-purpose structure field intended to store a limited amount of arbitrary custom data). Most of the known solutions either require significant manual proof overhead (e.g. VERIFAST [6]) or limit the analysis to memory safety (by analyzing only the shape of the data in the heap, but not the contents of non-pointer variables) and also allow imprecise analysis in certain cases in form of false alarms (e.g. SLAYER [7]). Despite those limitations these approaches quite efficiently handle most common operations on recursive data structures e.g. lists and trees and were successfully applied in practice for verification of low-level system code.

Approaches from the other category suggest a more naive modeling of memory state with a number of logical arrays (from the well-known and well-automated array theory) and leaving the problem of dealing with particular features of recursive data structures (e.g. formalizing the reachability relation) and memory separation (i.e. disjointness of memory areas) to the user. To compensate for the lack of built-in support for particular properties of recursive data structures the user is usually provided with an expressive logical framework supporting definition of general inductive predicates, recursive functions, axiomatic specifications and the ability to use appropriate proof techniques (e.g. proof assistants or lemma functions). These approaches trade relatively large specifications and possibly longer proofs for the expressiveness of the supported logic. They are implemented in many verification tools e.g. VCC [8,9], JESSIE [10] and WP plugin for FRAMA-C [11]. In this paper we are focusing on this second category of approaches. The ultimate purpose of our ongoing study is to demonstrate how it can be optimized and enhanced to support a simpler and more natural and concise specifications and possibly proofs for critical programs where reasoning about memory separation, recursive data structures as well as low-level memory representation of data is relevant. Linux kernel modules are especially good examples of such programs.

Since in this context the supported logical fragment is usually quite expressive, the corresponding decision problems are usually undecidable in general. Thus instead of providing a guarantee of decidability the corresponding verification tools usually implement various optimizations of the verification conditions (VCs) they produce based on relatively simple memory separation heuristics such as typed memory modeling [12], component-as-array modeling (also known as Burstall and Bornat memory model) [13, 14] and region-based memory separation [15]. We chose the region-based memory model as a basis of our further development. So the next section briefly presents the idea of region-based memory modeling and the subsequent sections refine and transform the idea to extend the expressiveness of the basic model for support of low-level pointer operations such as pointer type casts and pointer arithmetic beyond array indexing.

## 2 Overview

In Sect. 3 we first briefly introduce the notions of region-based memory modeling and separation analysis in the context of deductive verification tools (Subsect. 3.1). Then in the Subsect. 3.2 we turn our attention to the fact that most existing separation analyses are devised to assure soundness at the cost of losing precision. While this trade-off is well suitable for most of the traditional applications of separation analysis (e.g. optimizing compilers and automatic static checkers), in the context of deductive verification it's not that clearly justified. Even a relatively small change in separation assumptions, e.g. separation between different structure fields of the same type, can lead to changes in the performance of SMT-solvers on the resulting formulas up to an order of magnitude [9]. On the other hand, in the context of a deductive verification tool, that is always interactive, we don't need to ensure soundness of the separation analysis in the general case, but can verify its soundness separately for every individual code fragment by emitting the corresponding VCs. In case the VCs fail to verify the soundness of the analysis can be recovered by relying on user-supplied annotations. Thus we arrive at the notation of *user-guided separation analysis* that we introduce in Subsect. 3.3. In this subsection we also explain how typed memory model with reinterpretation [12] can be regard as an instance of user-guided separation analysis and how the corresponding solution can be extended to integrate user-guided region analysis, support for memory reinterpretation and memory safety checking into a single unified memory model. We introduce such a memory model in the following sections. In Sect. 4 we introduce a sufficiently expressive core language whose semantics closely follows that of C and is intended to be modeled by the suggested memory model. In Sect. 5 we provide an argument for the soundness of the proposed model and in Sect. 6 we identify the minimal set of necessary annotation capabilities that should be offered to the user to allow annotating any correct code fragment in such a way that its semantics is accurately represented in the corresponding model (thus also demonstrating completeness of the model). In Sect. 7 we provide some preliminary results on the amount of memory model-specific user annotations needed in practice for

verification of a real Linux kernel module. In Sect. 8 we briefly discuss current very basic support for framing and the corresponding directions of future work. We finally conclude in Sect. 9.

### 3 Region-Based Memory Modeling

#### 3.1 Basic Idea

Let’s demonstrate the basic idea of region-based memory modeling by a simple example. Suppose we are given an array `arr` of integers (`int`) and two pointers, `pf` and `p1`, referring to the first and the last element of the array `arr` correspondingly. Suppose also we have a pointer `pe` to an integer that is not an element of the array `arr`. Consider a single C decrement operator:

```
*pe--;
```

Suppose the operator is executed in a context where the following precondition is fulfilled:  $\sum_{i=0}^{p1-pf} pf[i] \geq 0 \wedge (pe < pf \vee pe > p1)$ . To formally prove a trivial postcondition  $\sum_{i=0}^{p1-pf} pf[i] \geq 0$  we can model the varying memory state naively using a single sequence of logical arrays  $M_i$  mapping the memory addresses to the corresponding integer values (let’s assume for a moment that we only have integer variables in memory). Then the sum can be modeled by an uninterpreted function  $S(p_f, p_l, M_i)$  with the necessary axiomatization. Thus using the strongest postcondition semantics [16] we obtain the following formula:

$$\left. \begin{array}{l}
 S(pf, p1, M_0) \geq 0 \wedge (pe < pf \vee pe > p1) \quad \textit{precondition} \\
 \wedge M_1 = M_0[pe \leftarrow M_0[pe] - 1] \quad \textit{*pe--}; \\
 \wedge (\forall M_i, M_j \in \mathbb{Z} \rightarrow \mathbb{Z}. \forall p_f, p_l \in \mathbb{Z}. \\
 (\forall x \in \mathbb{Z}. p_f \leq x \leq p_l \Rightarrow M_i[x] = M_j[x]) \\
 \implies S(p_f, p_l, M_i) = S(p_f, p_l, M_j)) \quad \textit{axiom} \\
 \wedge \neg(S(pf, p1, M_1) \geq 0) \quad \textit{negated postcondition.}
 \end{array} \right\}$$

The formula is a conjunction of the strongest postcondition of the given operator in the given context with the negated postcondition to be verified. The unsatisfiability of this conjunction implies inconsistency between any possible state of the program after execution of the given operator and any state where the postcondition in question is *not* fulfilled. Thus if the strongest postcondition itself is not inconsistent, the unsatisfiability of the formula implies the validity of the postcondition for any possible execution. The unsatisfiability of this formula can be determined by most modern SMT-solvers with support for quantifiers and theory of arrays, although satisfiability in first-order logic with arrays is undecidable in general. But the fact that the formula for even a trivial postcondition turned out to include five nested universal quantifiers demonstrates significant inefficiency of the chosen approach to modeling varying memory state.

The basic idea of region-based memory modeling is to rely on the results of a preliminary *separation analysis* that is able to separate pointer expressions in the program into disjoint sets so that any two pointers taken from different

sets necessarily (under certain assumptions) reference disjoint memory locations (here we only consider pointers and locations of simple types e.g. `int`). Such disjoint sets of pointer expressions are further referred to as *regions*. In our example such separation analysis could give us a hint that the pointer expressions `pf` and `p1` belong to the same region while the expression `pe` belongs to a different one. So this additional assumption can be used to encode the formula more efficiently by modeling the program state with two separate sequences of logical arrays:  $M_i^{\text{arr}}$  and  $M_j^{\text{pe}}$  (one sequence per region). Now the precondition stating disjointness of locations addressed by the sets of pointers  $\{\text{pf}, \dots, \text{p1}\}$  and  $\{\text{pe}\}$  is unnecessary. The formula becomes as small as:

$$S(\text{pf}, \text{p1}, M_0^{\text{arr}}) \geq 0 \wedge M_1^{\text{pe}} = M_0^{\text{pe}}[\text{pe} \leftarrow M_0^{\text{pe}}[\text{pe}] - 1] \wedge \neg(S(\text{pf}, \text{p1}, M_0^{\text{arr}}) \geq 0),$$

which is trivially unsatisfiable. Even though such optimization doesn't guarantee decidability in general, there are studies [9, 15] showing it can lead to significant benefits in practice.

### 3.2 Soundness/Precision Trade-Off

Now the problem with various fully automatic separation analyses (they are usually called alias analyses) in context of deductive verification is that most of them significantly loose precision in order to assure soundness. A typical example of this would be:

```
int n; char *pc = (char *)&n;
```

Here many existing separation analysis algorithms would assign the expressions `&n` and `pc` the same region. While this can be perfectly relevant for, say, an optimizing compiler, for a deductive verification tool such behavior is usually very undesirable. To understand this it's sufficient to consider an assignment `*pc = 0`; and its effect on the value of the variable `n`. Precise modeling of such assignments can significantly complicate the resulting formulas and effectively eliminate any benefit of using the separation analysis. A similar loss of precision typically occurs to a conventional separation analysis when it encounters a non-trivial pointer arithmetic or an integer-to-pointer type cast. This is one of the reasons why a special separation analysis for deductive verification was suggested in [15].

But the proposed solution was quite limited. To make the analysis sound many aspects of C were restricted. In particular, no nested structures, pointer type casts and unions were allowed. Essentially, all C constructs that can potentially violate separation between pointer expressions of different types were prohibited. The proposed solution was partially improved in [17], but many programming idioms widely used in the Linux kernel e.g. nested structures, the `container_of` macro (essentially equivalent to `(struct container *)((char *)p_containee - offsetof(struct container, containee))`) and non-prefix pointer type casts (e.g. `(char *)p_int`), were still unsupported by the separation analysis and the corresponding memory model.

### 3.3 User-Guided Separation Analysis

**General idea.** Yet in the context of a deductive verification tool there are very useful opportunities that were missed by the separation analysis suggested in [15]: it's the availability of user support provided through annotations and the possibility to ensure correctness of the region analysis for every particular code fragment separately by explicitly emitting the necessary VCs.

The availability of a *a posteriori* check of the region analysis results (by generating appropriate VCs) and the opportunity to require user annotations in some special cases suggest an essentially reversed trade-off for a separation analysis suitable for a deductive verification tool, i.e. an analysis that sacrifices soundness for better precision. In the context of a deductive verification tool we can safely use an *unsound* pointer analysis that initially makes unrealistically optimistic assumptions about memory separation. If the result of the analysis turns out to be incorrect for a particular code fragment, this can be detected by the corresponding VCs that in this case become invalid. The user is thus informed about a failed verification attempt due to unsoundness of the region analysis and is requested to provide additional annotations to guide the analysis and recover soundness.

Here an important downside of the suggested approach emerges. With this approach the user gets informed about a need for additional annotations to recover soundness, but does not necessary get a clear guidance about the particular problem and a concrete form of the annotations required. One of a few solutions to this problem is to make the region analysis simple and transparent for the user so the one can understand its limitations, manually investigate the particular reason for unsoundness and come up with the necessary additional annotations. This significantly limits the potential intricacy of the analysis involved, since its functioning should be easily traceable and the results should be easily predictable by the user. So primary benefit should come not from the elaborateness of the region analysis, but from the relevance of the initial optimistic assumptions and the rules applied by the analysis for the vast majority of real code fragments. Also the cases when these assumptions do not hold should be clearly distinguishable by the user.

Let's demonstrate how this general approach was earlier applied in a relatively simple memory model of the VCC2 deductive verification tool.

**Typed memory model with reinterpretation as user-guided separation analysis.** The solution involving additional annotations and VCs as appeared in VCC2 was called *typed memory model with reinterpretation*. The initial optimistic assumption it adopted was unconditional separation between memory areas addressed by pointers to different simple types. The technique applied for checking the validity of this assumption consisted in maintaining a special ghost (model) set of *typed pointers* with two available operations (*split* and *join*) and an invariant stating that a memory address must be referred to by no more than a single typed pointer in the set. Each pointer dereference was guarded with a check for containment of the dereferenced typed pointer in the ghost set. In case of a failing containment check, the two available model operations



(*split* and *join*) could be inserted by the user in the appropriate places to locally adjust the separation assumptions. The *split* operation allowed to convert a pointer addressing an arbitrary continuous object (variable, structure or array) into the corresponding set of pointers to `char`, while the *join* operation allowed to perform the reverse transformation. Instances of those two operations were called *memory reinterpretation*. With the use of reinterpretation the following code fragment unsupported by the basic region-based model ([15]) as well as the typed memory model in its initial form (without the use of reinterpretation):

```
int n = 0;
char *pc = (char *)&n, c;
c = *pc;
n = 1;
```

can be easily made admissible by inserting two user annotations corresponding to the operations on the ghost pointer set:

```
int n = 0;                                {(&n, int)}
char *pc = (char *)&n, c;                {(&n, int)}
/*@ split(&n); */                          {(&n + i, char) | 0 ≤ i < sizeof(int)}
c = *pc;                                   {(&n + i, char) | 0 ≤ i < sizeof(int)}
/*@ join(&n); */                            {(&n, int)}
n = 1;                                     {(&n, int)}
```

The state of the ghost typed pointer set after execution of each operator is shown on the right. The memory occupied by the variable `c` having no aliases is not indicated in the pointer set. As said above, in the typed memory model with reinterpretation all the indirect memory access operations (pointer dereferences and operations on possibly aliased variables) are supplied with the corresponding VCs that check for the presence of the corresponding typed pointer in the ghost pointer set at the current state, e.g. `*pc` implies  $\llbracket pc \rrbracket_{\mathcal{E}} \in \{(\&n + i, \text{char}) \mid 0 \leq i < \text{sizeof}(\text{int})\}$ , where  $\llbracket \cdot \rrbracket_{\mathcal{E}}$  denotes evaluation of the expression in the current state of the environment  $\mathcal{E}$  (here in the example  $\mathcal{E} = \{pc \mapsto \&n\}$ ). The corresponding VC is invalid if the first user annotation (*split*) is omitted. This corresponds to the fact that initial optimistic type-based separation assumption is not valid for this code fragment.

Thus despite the type-based separation assumption is generally wrong and makes the (trivial) separation analysis unsound, this unsoundness is necessarily detected by the failing verification attempts and can be eliminated by providing additional annotations. On the other hand, the introduction of type-based separation gave a significant boost to the VCC verification tool allowing, in particular, practical verification of some recursive data structures [12].

**Extending the approach with memory safety and regions.** The memory reinterpretation mechanism previously applied in typed memory model can be quite naturally extended to keep track of the currently allocated memory to support verification of memory safety properties. To do this one needs to augment the corresponding memory allocation and deallocation operations with the corresponding semantics regarding the state of the typed pointer set. Also, the initial state of the set should be chosen empty while at the locations correspond-

ing to the end of program execution the corresponding VCs should be generated ensuring the final emptiness of the set. Such extension allows to get rid of the need for additional separate memory safety model with its own annotations and VCs, but it also has its drawbacks.

In the initial typed model a failed verification attempt can quite clearly point to the particular problem of the separation analysis. With memory separation soundness and memory safety checks combined, the cases where separation analysis is unsound are not anymore clearly distinguishable from true memory safety alarms, incomplete preconditions (that should require the corresponding memory to be allocated) and, due to general undecidability of the underlying logic, also false alarms arising from incompleteness of the decision procedures. So even though the overall verification framework remains sound and the number of required VCs decreases (no need for separate memory safety checks), understanding the output of the verification tool becomes more complicated. Here again we rely mostly on the relative simplicity and predictability of the separation analysis.

We suggest extending the typed memory model with memory safety checking and interactive region separation based on relatively simple rules. The suggested model is also extended with component-as-array modeling of structure fields [14], but this extension was already proposed and implemented earlier in VCC3 and is described in paper [9].

So the next step that we suggest in this paper is to use the same ghost pointer set yet another time to ensure the soundness of region-based separation analysis. This can be done by refining the set of typed pointers ( $address, type$ ) into the set of pairs ( $address, region$ ) where regions are assigned to pointers according to the rules of region analysis. Here we implicitly assume the invariant restricting admissible region analysis algorithms to those that always place pointer expressions of different types into different regions. This implies that any region  $\rho$  always has a single corresponding type  $\tau(\rho)$ . If we denote the fact  $(a, \rho) \in V$  where  $V$  is the set of currently allocated pointer-region pairs as “the address  $a$  is allocated in region  $\rho$ ”, then the corresponding invariant on the pointer set transforms into the following statement: Any address can be allocated in at most one region. As explained further in Sect. 5, with this invariant the VCs ensuring the correct use of reinterpretation are sufficient to also guarantee the soundness of any region analysis satisfying type-based separation between regions (every region is assigned exactly one corresponding type).

Thus we suggest to use the set of pointer-region pairs simultaneously for three distinct purposes, namely: (1) Ensuring the correct use of reinterpretation mechanism; (2) Verifying memory safety properties; (3) Ensuring the soundness of region-based separation analysis. To refine more on this basic idea let’s introduce a sufficiently expressive core language capturing all the basic C language patterns involving pointers, structures, unions, arrays, pointer type casts and arbitrary combinations of those constructs.

## 4 The Core Language

Let's begin with a simplifying assumption. We assume all pointers to be pointers to structure types. Pointers to simple (non-composite i.e. arithmetic or pointer) types can be rewritten as pointers to special dummy structures of the form `struct t_s { t f; }`; where  $t$  is the simple type and  $f$  is an arbitrary field name. The dereferences of the form `*p` where  $p$  has type  $t$  are rewritten to `p->f`. This transformation is also described in [15].

But unlike the model suggested in the paper we don't apply nested structure elimination. Instead we suggest to make nested structures part of the memory model and support them directly by providing two corresponding core language constructs: `&p->f` and `container_of(p, f)` where `container_of` corresponds to the `container_of` macro (subtraction of the nested structure offset defined in Sect. 3.2) widely used in the Linux kernel. Let's consider motivation for introduction of explicit `&p->f` and `container_of(p, f)` constructs in more detail. Consider the following code fragment:

```
struct outer {
    struct inner { char c; } inner;
    int a; } outer;
struct inner *pinner = &outer->inner;
struct outer *pouter =
    container_of(pinner, struct outer, inner);
pouter->a = 0;
```

In the typed memory model (even without reinterpretation) the separation analysis for this code fragment is trivial since every pointer to a simple type addresses the memory object of that type, with initial optimistic type-based separation correctly assumed and no splits or joins required. So in typed memory model (and actually also in its extension with component-as-array modeling) there is no need in special support for `&outer->inner` or `container_of(pinner, struct outer, inner)` constructs. They can be simply reduced to combinations of more "primitive" pointer arithmetic and pointer type cast operations: `&outer->inner`  $\rightsquigarrow$  `(struct inner *)((char *)outer + offsetof(struct outer, inner))`, `container_of(pinner, struct outer, inner)`  $\rightsquigarrow$  `(struct outer *)((char *)pinner - offsetof(struct outer, inner))` (the latter is essentially just an expansion of the corresponding macro). It's important to note that in the typed memory model it's irrelevant whether `offsetof(struct outer, inner)` is a statically known constant, it can be replaced with any expression that evaluates to the same value without any consequence for separation analysis. Yet in the memory model with regions and memory safety checks this code fragment is non-trivial. To verify the validity of the pointer dereference `pouter->a` the pointer expression (variable) `pouter` should be necessarily assigned the same region as the expression `&outer`. But this can only be done if the corresponding offset `offsetof(struct outer, inner)` is a statically known constant. In general with the use of arbitrary expressions as offsets, the problem becomes undecidable and the region-based model either collapses

to the typed memory model or has to remain unsound at least for some uncommon uses of pointer arithmetic and pointer type casts. So according to the overall idea of a simple user-guided separation analysis a clear-cut criterion is introduced.  $\&p \rightarrow f$  and *container\_of*( $p, f$ ) become primitive constructs given the corresponding special semantics with respect to region analysis and the remaining corner cases generally result in unsoundness (of the analysis, but not the model) and has to be resolved by inserting a newly introduced specification construct *may\_alias*( $p_1, p_2$ ). This construct only affects region analysis and has no-op operational semantics and it's especially important for the completeness of the memory model addressed in Sect. 6. Since we suggest to support reinterpretation and integrate it with memory safety checks we need four corresponding core language constructs: *split*( $p$ ), *join*( $p$ ),  $p = \text{alloc}(n)$  and *free*( $p$ ). The allocation operator *alloc* is type-specialized (it's not an ordinary C function) and unlike the usual `malloc` function it allocates memory for  $n$  consecutive elements of the type corresponding to the pointer expression (lvalue)  $p$  in its left hand side. So the suggested model initially requires special treatment of allocation functions. However, the reinterpretation actually makes it possible to specify general allocation functions e.g. `malloc` without ad-hoc means. We don't use this approach initially to make the subset of programs supported without additional annotations (reinterpretation) more comprehensive.

Since we intend to define a core language of contract-free paths to be used in context of deductive verification we also need constructs for assumptions (at least at the start of a contract-free path), assertions (at least at the end of a contract-free path), and non-deterministic memory updates needed for support of framing contracts (specifying aggregated effects of function calls, more specifically, each call site consists of an assertion of the function's precondition, some non-deterministic memory updates, and an assumption of the function's postcondition). Currently we introduce three corresponding constructs: *assume*( $c$ ), *assert*( $c$ ) and *havoc*( $p, f_{pv}$ ). The metavariable  $c$  denotes any predicate in the appropriate specification language (can be a fragment of a real C specification language e.g. ACSL [18]). The suggested model naturally incorporates the well-known component-as-array modeling technique, i.e. the state of each structure field is modeled by a separate sequence of logical arrays. Let's further denote the region assigned by the separation analysis to a pointer expression  $p$  as  $P(p)$  and the corresponding sequence of memory states for a field  $f$  as  $M_i^{p \rightarrow f}$ . The operator *havoc*( $p, f_{pv}$ ) makes the entire state of the memory  $M^{p \rightarrow f}$  corresponding to the field  $f$  of the pointer expression  $p$  non-deterministic (this is easily modeled by increasing the current index  $i$  of the sequence  $M_i^{p \rightarrow f}$  without restricting the value of the corresponding fresh logical array).

Now the complete abstract syntax of the core language can be presented, as shown in Fig. 1. Besides only allowing pointers to structures here we make the second simplifying assumption that a single integral type of the same size as any pointer type is available. The variables occurring in the contract-free path are split into integer variables ( $v$ ) and pointer variables ( $p$ ), the terms and

$t_v ::=$   $i$   $p_1 - p_2$   $p_1 == p_2$   $p \rightarrow f_v$  $t_p ::=$   $\text{NULL}$   $p + v$   $p \rightarrow f_p$   $\&p \rightarrow f_s$   $\text{container\_of}(p, f_s)$	$o ::=$   $v = t_v$   $p = t_p$   $p = \text{alloc}(n)$   $\text{free}(p)$   $p \rightarrow f_v = v$   $p_1 \rightarrow f_p = p_2$   $\text{assert}(c)$   $\text{assume}(c)$   $\text{havoc}(p, f_{pv})$   $\text{split}(p)$   $\text{join}(p)$   $\text{may\_alias}(p_1, p_2)$	$s ::=$   $\varepsilon$   $o; s$
---	---	--

**Fig. 1.** Abstract syntax of the core language.

fields are split in the same way ( $t_v$ ,  $t_p$ ,  $f_v$ , and  $f_p$ ;  $f_{pv}$  is either  $f_p$  or  $f_v$ ). The fields corresponding to nested structures are denoted as  $f_s$ . The metavariable  $i$  denotes an arbitrary arithmetic expression (we don't describe modeling of arithmetic operations in this paper),  $o$  denotes core language operators and  $s$  denotes contract-free paths. The core language assumes a type system compatible with that of C and extended with a region environment  $P$  populated by the separation analysis. Since soundness is checked by emitting the corresponding VCs, we don't need any concrete typing rules for the environment  $P$ . Instead we need to supply the operations modifying the ghost pointer set  $V$  (***alloc***, ***free***, ***split*** and ***join***) with the semantics that preserves the necessary invariants (see Sect. 5).

#### 4.1 Translation of C

Now let's show that not only the constructs earlier supported by the model described in [15] and nested structures, but also arbitrary unions and pointer type casts can be expressed in the suggested core language. The idea is since the language supports nested structures, prefix pointer casts can be directly expressed by using the corresponding operators of the form  $p_i = \&p_o \rightarrow f$  and  $p_o = \text{container\_of}(p_i, f)$  where  $f$  is the first field of the outer structure referenced by  $p_o$  and  $f$  has the same type as pointer  $p_i$  (so the corresponding prefix casts are  $(\text{typeof}(p_i))(p_o)$  and  $(\text{typeof}(p_o))(p_i)$ ). Then to express non-prefix pointer casts and unions we introduce a special dummy structure type ***void*** with no fields and place a special field  $f_{s.\text{void}}$  of that structure type (a nested structure of zero size) as the first field to every structure type  $s$  other than the ***void*** itself. Thus a cast of the form  $(\text{struct } s_1)p$  where  $p$  has type  $\text{struct } s_2$  can be translated as  $\text{container\_of}(\&p \rightarrow f_{s_2.\text{void}}, f_{s_1.\text{void}})$ . In this setting unions can be treated as pointers to the ***void*** structure (the type ***void*** \* can also be treated this way). Then referring to a field of a union can be regarded as the corresponding pointer type cast. The only corner case arises from the use of nested unions and

arrays (when they are used as structure fields), but even those have to be treated specially only in the semantics of allocation and deallocation operators.

## 5 Soundness

We suggest demonstrating the soundness of the memory model for any arbitrary region analysis. We only assume that the results of the region analysis are represented by the region environment  $P$  mapping every syntactic pointer core language term ( $t_p$ ) of the target contract-free path to a corresponding region. We also assume, as mentioned earlier in Sect. 3.2, that every region  $\rho$  has a single corresponding structure type tag  $\tau(\rho)$ .

We introduce two semantics of the core language described in the previous section: one that closely corresponds to the operational semantics of the appropriate fragment of C and one that follows closely the semantics as formulated by the generated VCs. The former semantics is further denoted as *reference* semantics while the latter one is further called *model* semantics.

We can use small-step operational semantics [19] with non-deterministic choice to capture all possible executions of a contract-free path starting from a particular state using a single evaluation relation. We define intermediate evaluation state in the reference semantics as a quintuple  $(\mathcal{E}_p, \mathcal{E}_v, \mathcal{B}, \mathcal{V}, \mathcal{L})$ , where  $\mathcal{E}_p$  and  $\mathcal{E}_v$  denote the environments for pointer and integer variables correspondingly,  $\mathcal{B}$  represents the state of actual program memory (the map from addresses to the corresponding values),  $\mathcal{V}$  represents the set of currently allocated (*valid*) addresses (by mapping the addresses to the set  $\{\perp, \top\}$ ) and  $\mathcal{L} : \mathbb{B}^d \rightarrow \mathbb{N} \cup \{0\}$  represents the lengths of currently allocated blocks. The map  $\mathcal{L}$  is needed to formalize the semantics of the **free** operator to be close enough to the standard C function **free**. The set of values (terminal evaluation states) for contract-free paths is defined as  $\{\top, \perp\}$ , i.e. either any possible execution of the path respects all explicit and implicit specifications (**assert** operators and VCs for memory safety), which corresponds to the value  $\top$ , or there exists an execution of the path violating at least some specifications ( $\perp$ ).

An intermediate state in the model semantics is defined as a triple  $(\mathcal{E}_p, \mathcal{E}_v, \mathcal{M})$  where  $\mathcal{E}_p$  and  $\mathcal{E}_v$  are exactly the same as in reference semantics while the states of  $\mathcal{B}$ ,  $\mathcal{V}$  and  $\mathcal{L}$  are modeled with a set of maps  $\mathcal{M}$ . Each map in the set  $\mathcal{M}$  is uniquely identified by a pair  $(\rho, f)$  where  $\rho$  is a region and  $f$  is a corresponding structure field, so that  $p \rightarrow f$  is a valid lvalue as long as  $p$  has type  $\tau(\rho)$ . Besides normal structure fields  $f$  we also introduce two special fields  $s.v$  and  $s.l$  for every non-void structure  $s$ . With these special fields, the set of maps  $\mathcal{M}$  is able to model the reference maps  $\mathcal{V}$  and  $\mathcal{L}$  correspondingly. The most crucial difference between model and reference semantics is the separate modeling of different memory regions and different structure fields. Another difference is that in every map of the form  $\mathcal{M}_V^\rho$  uniquely identified by the region  $\rho$  and the field  $f_{\tau(\rho).v}$  the validity of the memory area corresponding to all non-composite fields of structure  $s$  at address  $a$  is represented by a single binary value  $\mathcal{M}_V^\rho[a]$ . This value also indicates whether the structure at  $a$  is currently allocated in the region  $\rho$ .

Let's summarize the invariants maintained by the reference and models semantics. These invariants determine constraints on the corresponding correct intermediate evaluation states.

Each reference evaluation state of the form  $(\mathcal{E}_p, \mathcal{E}_v, \mathcal{B}, \mathcal{V}, \mathcal{L})$  satisfies the following constraints:

$$\begin{aligned}
\forall a, i \in \mathbb{B}^d. i < \mathcal{L}[a] &\implies \mathcal{V}[a+i] && (\mathcal{L} \mapsto \mathcal{V}), \\
\forall a, i \in \mathbb{B}^d. 0 < i < \mathcal{L}[a] &\implies \mathcal{L}[a+i] = 0 && (\mathcal{L} \mapsto \mathcal{L}), \\
\forall a \in \mathbb{B}^d. \mathcal{V}[a] &\implies \exists i \in \mathbb{B}^d. \mathcal{L}[a-i] > i && (\mathcal{V} \mapsto \mathcal{L}), \\
\neg \mathcal{V}[0] &&& (\text{NULL}).
\end{aligned}$$

Here  $\mathbb{B}^d$  is the set of all memory addresses ( $d$  is the bit-length of an address e.g. 32 or 64). Operations  $+$  and  $-$  are wrap-around (assume arithmetic modulo  $2^d$ ).

In the model semantics the corresponding constraints get a bit more involved:

$$\begin{aligned}
\forall a, i \in \mathbb{B}^d, \rho \in \mathbb{R}. i < \mathcal{M}_{\mathcal{L}}^{\rho}[a] \times \text{sizeof}(\rho) &\implies \text{Valid}(a+i) && (\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{M}_{\mathcal{V}}), \\
\forall a \in \mathbb{B}^d, \rho \in \mathbb{R}. \mathcal{M}_{\mathcal{L}}^{\rho}[a] > 0 &\implies \text{uniq}_{\mathcal{L}}(a, \rho, \mathcal{M}_{\mathcal{L}}^{\rho}[a]) && (\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{M}_{\mathcal{L}}), \\
\forall a \in \mathbb{B}^d, \rho \in \mathbb{R}. \mathcal{M}_{\mathcal{V}}^{\rho}[a] &\implies && \\
\exists i \in \mathbb{B}^d, \rho' \in \mathbb{R}. \mathcal{M}_{\mathcal{L}}^{\rho'}[a-i] \times \text{sizeof}(\rho') > i &&& (\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{M}_{\mathcal{L}}), \\
\forall a \in \mathbb{B}^d, \rho \in \mathbb{R}. \mathcal{M}_{\mathcal{V}}^{\rho}[a] &\implies \text{uniq}_{\mathcal{V}}(a, \rho) && (\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{M}_{\mathcal{V}}), \\
\neg \text{Valid}(0) &&& (\text{NULL}^{\rho}),
\end{aligned}$$

where

$$\begin{aligned}
\text{Valid}(a) &\equiv \exists \rho \in \mathbb{R}. \text{Valid}^{\rho}(a, \rho), \\
\text{Valid}^{\rho}(a, \rho) &\equiv \exists f \in \mathbb{F}(\rho). \mathcal{M}_{\mathcal{V}}^{\rho}[a - \text{offsetof}(f)], \\
\text{uniq}_{\mathcal{L}}(a, \rho, l) &\equiv \\
\forall i \in \mathbb{B}^d, \rho' \in \mathbb{R}. i < l \times \text{sizeof}(\rho) \wedge (\rho' \neq \rho \vee i > 0) &\implies \mathcal{M}_{\mathcal{L}}^{\rho'}[a+i] = 0, \\
\text{uniq}_{\mathcal{V}}(a, \rho) &\equiv \left( \forall f \in \mathbb{F}(\rho), \rho' \in \mathbb{R}. \rho' \neq \rho \implies \neg \text{Valid}^{\rho'}(a + \text{offsetof}(f), \rho') \right) \wedge \\
&\quad \left( \forall i \in \mathbb{B}^d. 0 < i < \text{sizeof}(\rho) \implies \neg \mathcal{M}_{\mathcal{V}}^{\rho}[a+i] \right).
\end{aligned}$$

Here  $\mathbb{R} = \text{imgP}$  is the finite set of all regions assigned to pointer expressions by the region analysis,  $\mathbb{F}(\rho) = \text{fields}(\tau(\rho))$  is the set of all *non-composite* fields of the structure corresponding to the region  $\rho$ ,  $\text{sizeof}(\rho) = \text{sizeof}(\tau(\rho))$ .

The above constraints state that maps  $\mathcal{M}_{\mathcal{V}}^{\rho}$  and  $\mathcal{M}_{\mathcal{L}}^{\rho}$  should be self- and mutually consistent.  $\mathcal{L}$  and  $\mathcal{M}_{\mathcal{L}}$  should correctly specify the lengths of allocated blocks ( $\mathcal{L} \mapsto \mathcal{V}$  and  $\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{M}_{\mathcal{V}}$ ). Every valid address should be attributed to an allocated block ( $\mathcal{V} \mapsto \mathcal{L}$  and  $\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{M}_{\mathcal{L}}$ ). The blocks should be disjoint ( $\mathcal{L} \mapsto \mathcal{L}$  and  $\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{M}_{\mathcal{L}}$ ). In the model semantics every valid address is also attributed to a single valid structure as an address of one of its non-composite fields (see  $\mathcal{V} \mapsto \mathcal{M}_{\mathcal{V}}$  below). Valid structures should also be disjoint and every valid structure should be allocated in exactly one region ( $\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{M}_{\mathcal{V}}$ ). So the validity of structure in a region is represented by the value  $\top$  at the starting address of the structure in that region (as in definition of  $\text{Valid}^{\rho}$ ) and bottoms ( $\perp$ ) at all its other (non-zero) offsets in that region as well as all offsets of non-composite fields in all other regions ( $\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{M}_{\mathcal{V}}$ ). The validity of nested

structures is indicated in their corresponding regions. So the support for nested structures need not be manifested in the invariant.

The important difference between representations of validity and block lengths reflected in predicates  $uniq\mathcal{V}$  and  $uniq\mathcal{L}$  is that while validity of memory occupied by non-composite fields of nested structures is represented by the corresponding validity flags of the innermost structures (the immediate containers of the corresponding fields), the length of an allocated block is only represented once per block by the corresponding  $s.\mathcal{L}$  field of the first outermost structure in the block. This allows to apply arbitrary reinterpretations of the memory allocated within the block while still respecting the semantics of standard C functions `malloc` and `free` (or Linux kernel functions `kmalloc` and `kfree`) by ensuring that the whole block is deallocated simultaneously after passing its starting address to the function `free` (or `kfree`). The only extra limitation here is that before the call to `free` the layout (typed interpretation) of the memory should be restored to its original state as it was just after the allocation.

Now we can formulate the invariant relating the corresponding correct reference and model intermediate evaluation states:

$$\begin{aligned}
& (\forall a \in \mathbb{B}^d. \mathcal{V}[a] \implies Valid(a)) \wedge & (\mathcal{V} \mapsto \mathcal{M}_{\mathcal{V}}) \\
& (\forall a \in \mathbb{B}^d. \mathcal{L}[a] > 0 \implies \exists \rho \in \mathbb{R}. \mathcal{L}[a] = \mathcal{M}_{\mathcal{L}}^{\rho}[a] \times \mathit{sizeof}(\rho)) \wedge & (\mathcal{L} \mapsto \mathcal{M}_{\mathcal{L}}) \\
& (\forall a \in \mathbb{B}^d, \rho \in \mathbb{R}. \mathcal{M}_{\mathcal{L}}^{\rho}[a] > 0 \implies \mathcal{L}[a] = \mathcal{M}_{\mathcal{L}}^{\rho}[a] \times \mathit{sizeof}(\rho)) \wedge & (\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{L}) \\
\\
& \forall a \in \mathbb{B}^d, \rho \in \mathbb{R}. \mathcal{M}_{\mathcal{V}}^{\rho}[a] \implies \\
& \quad (\forall f \in \mathbb{F}(\rho). \\
& \quad \quad \mathcal{V}[a + \mathit{offsetof}(f)] \wedge & (\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{V}) \\
& \quad \quad \mathcal{M}_f^{\rho}[a] = \mathcal{B}[a + \mathit{offsetof}(f)]) & (\mathcal{M}_{\mathcal{B}} \stackrel{\leftarrow}{\mapsto} \mathcal{B}).
\end{aligned}$$

This invariant essentially establishes a bijection between the three maps  $\mathcal{V}$ ,  $\mathcal{B}$ ,  $\mathcal{L}$  and the map  $\mathcal{M}$ . ( $\mathcal{V} \mapsto \mathcal{M}_{\mathcal{V}}$ ) guarantees that any valid (currently allocated) address is valid in at least one region as an address of some non-composite structure field (a field of primitive type). ( $\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{V}$ ) states that validity of a structure in a region implies validity of all addresses corresponding to its non-composite fields and ( $\mathcal{M}_{\mathcal{B}} \stackrel{\leftarrow}{\mapsto} \mathcal{B}$ ) also states that this implies equivalence between the values in the maps  $\mathcal{M}$  and  $\mathcal{B}$  for those addresses. ( $\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{L}$ ) and ( $\mathcal{L} \mapsto \mathcal{M}_{\mathcal{L}}$ ) together with the consistency invariants ( $\mathcal{L} \mapsto \mathcal{L}$ ) and ( $\mathcal{M}_{\mathcal{L}} \mapsto \mathcal{M}_{\mathcal{L}}$ ) stated above provide one-to-one correspondence between the map  $\mathcal{L}$  and the corresponding set of maps  $\mathcal{M}_{\mathcal{L}}^{\rho}$ .

With all the self- and mutual consistency invariants maintained, soundness of the modeling can be ensured by simply checking the precondition  $\mathcal{M}_{\mathcal{V}}^{\rho}[\mathcal{E}_p(p)]$  for every pointer dereference of the form  $p \rightarrow f$ . These checks guarantee that the read and write accesses always happen to the maps  $\mathcal{M}_f^{\rho}$  that exactly represent the corresponding values from the map  $\mathcal{B}$ . The only remaining concern is to ensure the semantics of memory (de)allocation and reinterpretation operations (*alloc*, *free*, *split* and *join*) also respect the invariants. The corresponding formal description of their semantics is quite long, but essentially uninteresting, so we omit it in this paper.



If we denote the above invariants as  $(\mathcal{B}, \mathcal{V}, \mathcal{L}) \sim \mathcal{M}$ , the evaluation relation of the reference semantics as  $\blacktriangleright$ , and the evaluation relation of the model semantics as  $\triangleright$ , then the final soundness result can be presented as follows:

$$\forall \mathcal{E}_v, \mathcal{E}_p, \mathcal{B}, \mathcal{V}, \mathcal{L}, \mathcal{M}, s. (\mathcal{B}, \mathcal{V}, \mathcal{L}) \sim \mathcal{M} \Rightarrow s|(\mathcal{E}_v, \mathcal{E}_p, \mathcal{B}, \mathcal{V}, \mathcal{L}) \blacktriangleright^* \perp \Rightarrow s|(\mathcal{E}_v, \mathcal{E}_p, \mathcal{M}) \triangleright^* \perp.$$

This is a typical statement of a sound approximation: Provided the initial states respect the invariant, the correctness of the model guarantees the correctness of the original path. It's worth noting here again that the approximation remains sound regardless of the particular assignment of regions to pointer expressions (P).

Yet soundness only guarantees that an incorrect contract-free path (that may evaluate to  $\perp$  under the reference semantics) also remains incorrect (has  $\perp$  as one of its values) under the model semantics. This does not imply any guarantees about correct contract-free paths. Thus arises the completeness issue since the memory model doesn't impose any restrictions on the region analysis and an unrestricted analysis can easily assign the regions in such a way that makes any correctness under the model semantics impossible. For an example, a region analysis that assigns a fresh region to every pointer expression will make it impossible to correctly dereference any pointer.

## 6 Completeness

Since the overall approach of user-guided region analysis relies on the use of auxiliary annotations, the goal of this section is to identify the minimal required set of annotation constructs that allows the user to adjust any correct contract-free path without changing its reference semantics so that the path becomes correct (necessarily evaluates to  $\top$ ) under the model semantics. In this section we still impose only just a few additional restrictions on the kind of region analyses we admit. So instead of identifying the annotations potentially needed for a particular region analysis algorithm (or a restricted set of algorithms) we suggest the set of constructs and the corresponding annotation strategy that cover the worst case and allow to annotate any correct path for any admissible region analysis algorithm. This strategy is of course very redundant and not practical, but it proves the theoretical completeness of the proposed memory model and also provides the user with a guiding hint about what annotations should be inserted in practice for the cases when default optimistic assumptions of the region analysis end up being wrong for a particular code fragment. The actual amount of annotations really needed in practice is preliminary assessed in Sect. 7.

So there are essentially three reasons for the incompleteness of our region-based model. The first two reasons are concerned with separation assumptions, namely spurious (overoptimistic) separation between two regions of the same structure type and two regions of different structure types. The third reason is coarser granularity in representation of maps  $\mathcal{M}_V$  and  $\mathcal{M}_L$  relative to the maps  $\mathcal{V}$  and  $\mathcal{L}$ .

The first issue, spurious separation between regions of the same type can be easily addressed by explicitly providing the necessary equality constraints in the form of special unification operators *may\_alias* introduced in Sect. 4. These operators with no-op reference semantics can be added to the path in arbitrary places to constrain the separation analysis when needed. There is a subtlety here in the fact that an entirely unrestricted separation analysis can in theory assign different regions to different occurrences of the same expression (physical vs. structural equivalence between pointer expressions is irrelevant for soundness and so both are theoretically admissible) and also that the semantics of some operators (e.g. *alloc*) implicitly involves regions that can even have no corresponding pointer expressions syntactically present in the path (e.g. regions of the nested structures). So the separation analysis is further restricted to be control-flow insensitive and the existence of a map  $N$  from pairs of the form  $(\rho, f_s)$  to the corresponding nested structure regions is assumed. Since some operators also implicitly involve container regions (*split* and *join*, when casting the pointer to/from `char *`) the existence of a similar container structure map  $N^{-1}$  is also assumed. But the consistency of the two maps is, strictly speaking, not necessary for completeness since the number of regions is always final and bounded in advance and so necessary unifications can in theory be added explicitly with *may\_alias*. There is yet another issue with the presented core language since it doesn't allow arbitrary complex pointer expressions: the auxiliary variables have to be introduced. To resolve this, mandatory region unification on all assignment operators is required, which makes results of separation analysis invariant under the introduction of intermediate variables.

The second issue, spurious separation between regions of different types, has the solution already built into the core language, which is its support for reinterpretation (operators *split* and *join*). Since each operator (see  $o$  in Fig. 1) allows only a single pointer dereference, the same memory cannot be accessed through two pointers of different types in one operator. So in theory it's enough to add a *join* before each operator involving a pointer dereference and a *split* afterwards. This is completely analogous to the trick used in the completeness proof for the typed memory model in [12].

Now the most problematic concern for the incompleteness of the region-based memory model is that the reference semantics allows even only a part of a structure to be correctly allocated and successfully accessed as long as the operations only involve the allocated parts. In contrast, the invariants of the correct intermediate model evaluation states make some configurations of the map  $\mathcal{V}$  inexpressible within the corresponding set of maps  $\mathcal{M}_{\mathcal{V}}^e$ . In the model semantics a partially valid structure having both some valid and some invalid *non-composite* fields simply can not be represented (as stated by the clause  $(\mathcal{M}_{\mathcal{V}} \mapsto \mathcal{V})$  of the invariants  $\sim$  in Sect. 5). The issue also occurs when trying to translate a C program involving an addressing operator  $\&$  on a non-composite structure field. The problem is that to our knowledge there's no good known solution to resolve this issue in a way that is both sound and modular (that is also pointed out in [9]). We choose a non-modular solution which is to translate

the original program differently depending on which structure fields of simple types are addressed (either with `&` or by using `offsetof` corresponding to the field). In relation to the completeness of the model semantics in theory we can translate each non-composite field as a nested structure of the corresponding dummy structure type with a single field mentioned in Sect. 4. The operators on the path should be transformed correspondingly.

Thus we introduce three additional specification constructs, namely *may\_alias*, *split* and *join* and also suggest special preliminary code transformation for addressed structure fields of primitive types, where the addressing operator `&` can also be regarded as an annotation in the original program before the transformation. So by combining the suggested solutions together it's possible to transform any original path in such a way that whenever it's only related to the correct result ( $\top$ ) under the reference semantics, the result of its evaluation in model semantics is also necessarily  $\top$  i.e. in theory it's possible to demonstrate the correctness of any given path under the model semantics. Since the transformations don't change the reference semantics of the original path, the presented model can in theory be used to soundly prove correctness (evaluation to  $\top$  for any possible execution) of arbitrary individual contract-free paths.

## 7 Annotation Overhead

The worst-case annotation strategy suggested in the previous section assumes a very large potential number of annotations, exceeding the number of operators in the original code. However, the model can be easily viewed as a generalization of the model presented in [15], which was successfully applied to code fragments amounting to  $\approx 3000$  LoC from an embedded system for avionics. The model didn't make use of any additional annotations. To make the separation analysis a direct extension of the one considered in [15], it's enough to impose an additional constraint—the existence of a map  $\Delta$  from pairs of the form  $(\rho, f_p)$  to the corresponding regions of pointer expressions  $p \rightarrow f_p$  where  $p$  is assigned the region  $\rho$ . This constraint alone makes all the programs admitted by the basic region-based memory model [15] with its separation analysis also admissible by the model presented in this paper. But since the goal of the suggested model is support of C language constructs and programming idioms used in the Linux kernel, it's also reasonable to consider automatic (annotation-free) support of a larger language subset. Indeed it seems natural to show that the suggested model also supports those uses of arbitrarily nested structures (and arrays of structures) that are syntactically recognizable, since the constructions `&p → fs` and `container_of(p, fs)` are part of the core language. However, a rigorous proof of the corresponding statement requires introduction of quite a few new notions (e.g. syntactically recognizable use of nested structures) and is quite involved.

So far we only refer to our practical experience based on a custom Linux security module being approximately 3.5 KLoC that showed only 12 out of 255 functions needed any additional annotations required by the memory model (`to_int`, `of_int`, `may_alias` or explicit field addressing `&` absent in the original

code) totally amounting to 32 additional annotations. 6 of the functions requiring additional annotations involved memory reinterpretation between integral types of different size and byte reordering (those required 4 reinterpretation annotations for each function), 4 functions involved allocating memory for flexible array members (this is not supported automatically and requires reinterpretation annotations, also the semantics of allocation operator can be extended to support final flexible arrays), 1 function involved memory reinterpretation between a structure type and an array of *unsigned char* (the function required 2 annotations), and 1 remaining function involved a pointer type cast of the form  $(t **)p$  where  $p$  has type *void \*\** and  $t$  is not *void*, which is also considered reinterpretation by the model since *void \** and  $t *$  are treated as different primitive types (the function required 2 annotations). The implementation of the suggested memory model is not yet finished, primarily regarding framing and support for inductive and recursive logic definitions, so some additional kinds of annotations can be further required. But they are more likely to be related to framing and logic function domains rather than the memory model for contract-free paths.

## 8 Framing

The particular problem regarding embedding of a callee frame condition (contract) into the caller sequence of operators is the requirement to maintain the invariants underlying the soundness of the memory model. We suggest quite a simple solution that is to require the regions of the callee function to match the regions of the caller function precisely. This simple solution can potentially make the memory model significantly less efficient in some cases. Similar to the basic model [15] our implementation propagates unification i. e. equivalence of regions from the callee functions to the caller. This makes the region separation in higher-level functions more coarse-grained, but allows to support a significant subset of C without requiring additional annotations. The only case that is not covered by this solution is a call of a function which assumes more fine-grained separation that is already assumed in the caller function. This problem can be solved in a non-modular way by inserting the appropriate *may\_alias* annotation in the body of the callee function (this is what we currently use in practice). But this may potentially degrade the performance of SMT solvers on the callee function. So since this solution is not only non-modular, but also potentially disruptive, we suggest an alternative in the form of an additional specification construct allowing to move a set of allocated memory addresses between regions. This, however, requires formal introduction of address sets, corresponding operations, and mechanisms to compensate for the additional overhead arising from the use of such moving annotation. The development of the corresponding solutions is left for future work.

## 9 Conclusion

In this paper we suggested a modification of region-based memory model for deductive verification that allowed to support arbitrarily nested structures, unions and arrays, arbitrary pointer arithmetic and general pointer type casts. The support of nested structures and arrays is fully automatic and requires no additional annotation overhead, while the support for pointer arithmetic, unions and pointer type casts requires user annotations provided through three special specification constructs (*split*, *join* and *may\_alias*). Preliminary experimental evaluation on an industrial security kernel module showed a small required additional specification overhead related to the memory model (32 annotations per approximately 3.5 KLoC). Framing is not fully automatic and may require additional non-modular annotations that can potentially make the resulting SMT-formulas harder to decide. Overcoming this limitation is currently left for future work.

## References

1. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, LICS 2002, Washington, DC, USA, pp. 55–74. IEEE Computer Society (2002)
2. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic with trees and data. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 711–728. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-08867-9\\_47](https://doi.org/10.1007/978-3-319-08867-9_47)
3. Piskac, R., Wies, T., Zufferey, D.: Automating separation logic using SMT. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 773–789. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-39799-8\\_54](https://doi.org/10.1007/978-3-642-39799-8_54)
4. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. In: Barthe, G. (ed.) ESOP 2011. LNCS, vol. 6602, pp. 439–458. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-19718-5\\_23](https://doi.org/10.1007/978-3-642-19718-5_23)
5. Lahiri, S., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, pp. 171–182. ACM, New York (2008)
6. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-20398-5\\_4](https://doi.org/10.1007/978-3-642-20398-5_4)
7. Berdine, J., Cook, B., Ishtiaq, S.: SLAYER: memory safety for systems-level code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_15](https://doi.org/10.1007/978-3-642-22110-1_15)
8. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03359-9\\_2](https://doi.org/10.1007/978-3-642-03359-9_2)

9. Böhme, S., Moskal, M.: Heaps and data structures: a challenge for automated provers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 177–191. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22438-6\\_15](https://doi.org/10.1007/978-3-642-22438-6_15)
10. Moy, Y.: Automatic modular static safety checking for C programs. Ph.D. thesis, Université Paris-Sud, January 2009
11. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33826-7\\_16](https://doi.org/10.1007/978-3-642-33826-7_16)
12. Cohen, E., Moskal, M., Tobies, S., Schulte, W.: A precise yet efficient memory model for C. *Electron. Notes Theor. Comput. Sci.* **254**, 85–103 (2009)
13. Burstall, R.M.: Some techniques for proving correctness of programs which alter data structures. *Mach. Intell.* **7**(23–50), 3 (1972)
14. Bornat, R.: Proving pointer programs in Hoare Logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000). [https://doi.org/10.1007/10722010\\_8](https://doi.org/10.1007/10722010_8)
15. Hubert, T., Marché, C.: Separation analysis for deductive verification. In: Heap Analysis and Verification (HAV 2007), Braga, Portugal, pp. 81–93, March 2007
16. Dijkstra, E.W., Schölten, C.S.: The strongest postcondition. In: Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science. Springer, New York (1990). [https://doi.org/10.1007/978-1-4612-3228-5\\_12](https://doi.org/10.1007/978-1-4612-3228-5_12)
17. Moy, Y.: Union and cast in deductive verification. In: Proceedings of the C/C++ Verification Workshop, vol. Technical Report ICIS-R07015, pp. 1–16. Radboud University Nijmegen, July 2007
18. Baudin, P., Cuoq, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. Version 1.11 – Aluminium-20160501, September 2016
19. Plotkin, G.D.: A structural approach to operational semantics (1981)

# Indexing of Hierarchically Organized Spatial-Temporal Data Using Dynamic Regular Octrees

Sergey Morozov<sup>1,2</sup>, Vitaly Semenov<sup>1,3</sup>, Oleg Tarlapan<sup>1,2</sup>,  
and Vladislav Zolotov<sup>1</sup>(✉)

<sup>1</sup> Institute for System Programming, Russian Academy of Sciences,  
Moscow, Russia

{serg,sem,oleg,vladislav.zolotov}@ispras.ru

<sup>2</sup> Lomonosov Moscow State University, Moscow, Russia

<sup>3</sup> Moscow Institute of Physics and Technology, Dolgoprudny, Moscow Region, Russia

**Abstract.** The paper is devoted to theoretical and experimental study of indexing methods as applied to spatial-temporal datasets appearing in different science and industry domains. For this purpose a general spatial-temporal data model is presented as a scene that admits hierarchically organized, heterogeneous spatial objects with individual temporal behaviors. For the model presented we argue the relevance of dynamic event-driven regular octrees as an underlying spatial-temporal indexing structure to a wide class of applications and prove its effectiveness for queries such as scene reconstruction, region search, and collision detection.

For hierarchically organized scenes a complementary generalization of the octrees is proposed. Its performance and memory consumption advantages over traditional structures are confirmed by carrying out a series of computational experiments with industry meaningful datasets originated from the construction modeling applications. Results of computational experiments substantiate theoretical conclusions and demonstrate possibilities of creating efficient applications under the conditions of permanently growing scales and complexity of spatial-temporal data.

## 1 Introduction

Spatial-temporal information systems and, particularly, spatial-temporal database management systems, are becoming increasingly important in different science and industry domains such as agriculture, climatology, ecology, economics, telecommunication, transportation, navigation, construction, multimedia where real-world objects live in three-dimensional space and undergo to permanent changes [1, 2]. Traditional relational and object-oriented database technologies are not suitable for effective managing and retrieving such data. On the other side, research efforts in multidimensional databases showed that such queries cannot be efficiently resolved without considering the data semantics. Particularly, spatial, temporal, and spatial-temporal (e.g. velocity, acceleration) concepts must be captured to meet the performance requirements [3]. It becomes

especially important if computationally intensive queries like dynamic collision detection (finding all the spatial objects which are overlapped over a given time period) or motion planning (finding a conflict-free path in complex environment from source to destination position) must be resolved in a real-time application [4]. However, a lack of comprehensive spatial-temporal data models and holistic indexing frameworks prevents the creation of efficient applications.

A choice of spatial-temporal model plays a crucial role for the developed applications. But often this factor is neglected when the application is being designed and proper indexing structures and techniques are being selected. Spatial-temporal models can be classified into the following ten categories: Snapshot model, Space-Time Composite model, Simple Time-Stamping models, Event-Oriented models, Three-Domain model, History Graph model, Spatio-Temporal Entity-Relationship model, Object-Relationship model, Spatio-Temporal Object-Oriented models and Moving Object Data model. For brevity we omit the detailed comparative analysis of the models, addressing to [5,6].

Since the late 1980s different spatial-temporal indexing methods have been proposed and developed mostly as extension or generalization of well-known spatial data access structures. Following the excellent surveys [7,8], these structures can be roughly subdivided into three families:

- structures implying the space decomposition (overlapping linear quadtree, PMR quadtree);
- structures employing the object composition based on the R-trees (MR-tree, HR-tree, HR+-tree, 3-dimensional R-tree, 2+3 R-tree, MV3R-tree, 2-3 TR-tree, LUR-tree, RT-tree, STR-tree, TB-tree, SETI, SEB-tree, TPR-tree, PR-tree, VCI R-tree, STAR-tree, TPR\*-tree, REXP-tree, NSI, RST-tree);
- structures representing space and time using transformations (duality transformations, transformations with kinetic data structure, transformations with starting location and velocity).

As can be seen, less attention is paid to the methods based on regular and irregular space decomposition, although many structures like quadtrees, octrees, k-d-trees, puzzle-trees, X-Y-trees, BSP-trees, treemaps, multilevel grids, and metric trees are successfully employed in sophisticated applications of computer graphics, virtual and augmented reality, CAD/CAM. In previous studies we analyzed main families of spatial decomposition methods and singled out a family of indexing methods based on dynamic regular octrees [9]. Main advantage of regular octrees is cheap updates owing to a priori known position of cutting planes and a capability to meet permanent changes occurring in time. In combination with the temporal access methods, particularly those that rely on event indexing trees, regular octrees support spatial queries highly effective. Although non-uniformly distributed spatial data may result in unbalancing of indexes and some temporal queries need spanning the entire event history, these shortcomings do not prevent high performance results in some applications [10].

In the paper we present own Scene Model that incorporates different concepts of the spatial-temporal models mentioned above and leverages their constructive features. The model admits hierarchically organized, heterogeneous spatial



objects with individual temporal behaviors which combine both discrete events and continuous movements. For the model presented we argue the relevance of dynamic event-driven regular octrees as an underlying spatial-temporal indexing structure to a wide class of applications and prove its effectiveness for typical queries such as scene reconstruction, region search, and collision detection. The presence of hierarchical organization brings uniqueness in the model and results in a wider interpretation of queries addressed simultaneously to both parent and child objects between which the composition relations are established. For such cases we provide a hierarchical generalization of the proposed indices and confirm its performance and memory consumption advantages over traditional structures by carrying out a series of computational experiments with industry meaningful data sets originated from the construction modeling applications.

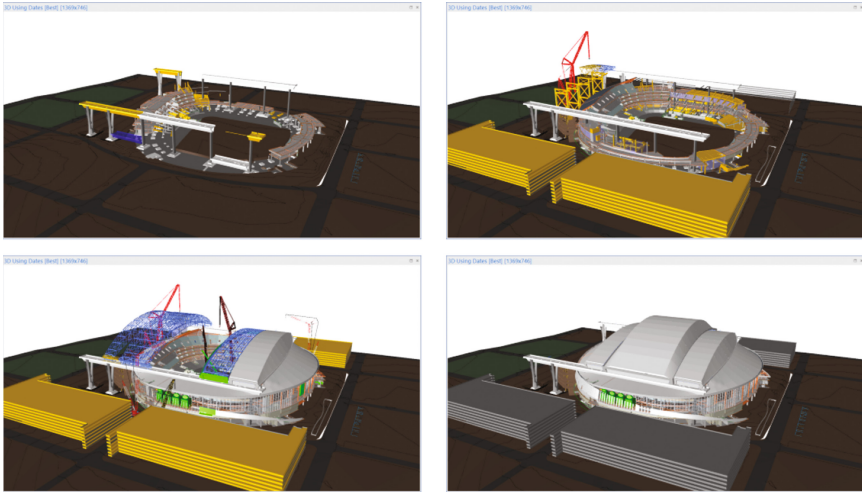
The rest of the paper is organized as follows. In Sect. 2 we present the scene model and focus on its peculiarities caused by a hierarchical organization of the spatial-temporal data and a wider interpretation of typical queries. Dynamic event-driven regular octrees are considered in Sect. 3 where their hierarchical generalization for the scene data is provided too. Section 4 is dedicated to theoretical complexity analysis of the data access methods based on the presented indexing structures. Results of computational experiments are discussed in Sect. 5. In Conclusions we summarize the achieved benefits of the proposed indexing methods and outline the directions of further research and the perspectives for their practical employment.

## 2 Scene Data Model

In the scene model the entire dataset is considered as a collection of dynamic spatial objects or as a scene. The scene consists of simple and compound objects which in turn consist of other children objects and, thereby, the scene has a hierarchical structure due to parent-child composition relations among objects. Simple objects may be points, lines, surfaces, volumes and hyper-volumes in high-dimensional spaces of corresponding geometry and topology semantics.

The composition relations introduced in the scene model have another assignment implying that spatial positions of the child object are always defined in a local coordinate system of its direct parent. That is true, except for the top-level objects which are located in a global coordinate system of the entire scene. With local positions, absolute position of any object in the global coordinate system can be easily computed by traversing all the parents and multiplying local transformation matrices as it is usually done in computer graphics applications. Bounding volumes like AABB (Axis-Aligned Bounding Box) (sometimes, they are also called Minimum Bounding Rectangles (MBR)) can be determined for any scene object in a local coordinate system of its parent.

A temporal behavior of the scene is modeled by means of defining and sequencing object events. Each event is associated strongly with one object and stores one timestamp when the event happens in the modeled reality. The scene model is a valid time model assuming that all the timestamps fix the absolute



**Fig. 1.** A series of images illustrating the construction project progress within the scene data model.

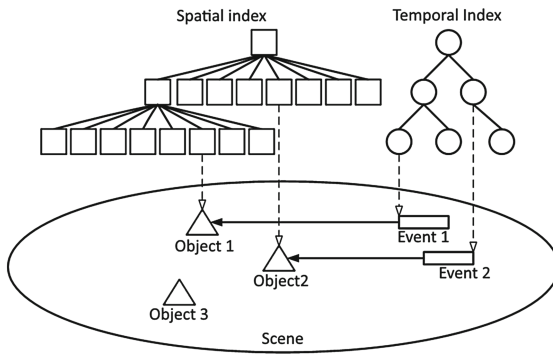
linear time points when the facts are true. Current modeling time is called a focus time. No transaction times peculiar to some other models are supported. The events are interpreted in a wider context as deterministic precedents which change the object states or modify their behavioral patterns. The considered events may be either appearance or disappearance of an object or a movement to another position along a given path. A sequence of all the events associated with a given object is called the object behavior and can include its multiple discrete-in-time appearances in the scene, its multiple disappearances from the scene as well as continuous-in-time movements. The events are defined so that any of their sequences is interpreted unambiguously, e.g. the existence of the object in the scene, its position and orientation can be determined as a function of time. The events driving continuous movements induce the subsequences of derived events which are not persistent data, but are involved in evaluation of temporal queries. During query processing the derived events are generated with a time step adaptively selected in accordance with the spatial-temporal coherence of the scene.

Thus, the scene model admits hierarchically organized, heterogeneous spatial objects with assigned individual behaviors combining both discrete events and continuous movements. The model has a lot of sophisticated applications in different sciences and industries. Figure 1 provides an application example aimed at modeling of construction projects and detecting spatial-temporal conflicts in the project schedules. A series of images reproduces a project progress on the construction site of a modern-of-art stadium. Building objects appear in the model scene at predetermined discrete times according to the project schedule, and remain there until the end of the entire modeling period. The objects

are placed in the positions and orientations of the relevant project 3D design documentation. Some technological objects, being installed, are later removed. Complicated construction operations are modeled using continuous movements.

### 3 Scene Indexing Structures

The discussed scene indexing structure combines an event tree and spatial decomposition trees. The event tree plays the role of primary temporal index which can be computed once and then be updated if only the model events have happened. It can be implemented as an AVL binary search tree or B-tree ordered by the event timestamps and identifiers. It allows fast lookup, efficient retrieval of events in a given time interval and quick updates when registering new events in the model. All these operations take time proportional to logarithm of the number of model events and this fact makes possible the use of the event tree for various computational tasks and evaluation of various temporal queries. One of the typical queries is the reconstruction of a dynamic scene at a given focus time.



**Fig. 2.** Spatial-temporal indexing structure for the one-level scene model.

Spatial decomposition trees are secondary indexes which are computed or updated whenever a model focus time is changed. They depend on the entire model dataset, and that is why their updates should be organized effectively. Our previous research has proved that regular octrees being used as spatial indexes exhibit high performance for such queries as frustum culling, nearest neighbor search, collision localization, hidden surface removal meanwhile being applicable to objects with extended borders. At the same time non-expensive incremental updates are admitted to bring the indexes to consistent states. Figure 2 illustrates the discussed indexing structure.

The traditional regular octrees use recursive subdivision of the scene space by planes perpendicular to coordinate axes into eight equal parts. This procedure is repeated recursively until the number of objects in each newly obtained octant

becomes less than some threshold  $m > 1$ . The resulting spatial decomposition is associated with a tree data structure. The tree root corresponds to the original AABB parallelepiped containing the complete dataset, and the vertices correspond to the nested octants that group objects on different hierarchical levels of spatial decomposition.

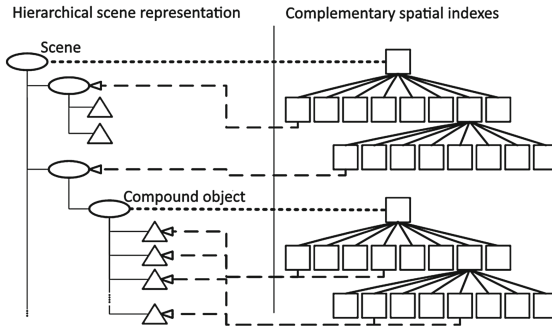
Previous theoretical and practical research showed that the presented indexing structure works well for both synthetic scenes and datasets originated from industrial applications under the condition that the scenes consist of simple objects and spatial-temporal queries are directly addressed to them. But the indexing structure becomes ineffective and shows degradable performance for the queries addressed simultaneously to both simple and compound objects of the hierarchically organized scenes. Indeed, traditional octrees do not reflect the composition relations among spatial objects while typical queries like region search and collision detection take other meaningful interpretations.

In the case of hierarchical scenes the region search query can be reinterpreted as a selection of the highest level objects which are placed inside the given region entirely or cross it partially. It does not make sense to include child objects in the resulting list once their high-level parents have been already identified and included. For example, one has to find all the equipment units and their pieces which are placed within some area of a construction site at a given focus time. Complex equipment, such as a crane or an excavator, may consist of thousands and millions of pieces. Using traditional octrees each piece should be indexed individually and then be analyzed and grouped with other localized pieces so that the resulting high-level objects can be identified. This process may consume significant computational resources, ultimately neglecting benefits from spatial indexing.

For hierarchical scenes the collision detection query should be reinterpreted too. The query aims at the determination of collisions between those simple objects which belong to different compound objects. For this purpose the top-level scene objects or preliminary selected objects are usually analyzed. Returning to our example, it is interesting to know which of equipment units and building elements intersect each other. Collisions of separate pieces of the same crane or excavator are beyond of the problem of modeling construction project progress. The use of traditional octrees for the query evaluation would lead to unreasonably high expenses. Indeed, it would require deploying the octree for all simple objects, determining all the objects collisions and then deciding which of the intersected objects belong to different equipment units and building elements. It is obvious that the most procedures are redundant and can be avoided by evolving the underlying indexing structure.

To address the discussed queries, we propose to employ the original scene representation and to deploy multiple octrees for those compound objects of the scene hierarchy which contain a relatively large number of child objects, including the scene as a top-level compound object. We call this indexing structure a complementary generalization of octrees meaning that it complements the original hierarchical scene representation. Figure 3 demonstrates an example of the

complementary generalization of octrees deployed for some compound objects of the presented scene. On the assumption that compound objects group child objects located close to each other, it can be argued that the complementary indexing structure leverages both spatial decomposition and object composition techniques. Being combined with the event tree, it can be used constructively to resolve complex spatial-temporal queries discussed above.



**Fig. 3.** Indexing structure for the multi-level scene model.

Consider the region search query performed in a dynamic hierarchical scene at a given focus time. It can be evaluated by recursive traversing through the scene hierarchy in a top-down manner while the visited objects are located within the region. Once the current object is located within the region entirely, it is added to the list of results, thereby preventing a redundant analysis of its children which may consist of thousands and millions of pieces. If the object is beyond the region, the object and its children are immediately avoided from further analysis. To identify whether a visited object is within the region or beyond the region, AABB of the object is checked. If the object crosses the region then the query readdresses to its children. To optimize this procedure for the traversed children, the complementary octree associated with the parent is utilized for the search propagation. If the complementary octree is not deployed for the parent by some reasons, then all its children are checked directly.

The collision detection query can be also evaluated by recursive traversing the scene hierarchy, but starting at the preliminary selected objects. In some sense it repeats well-known Bounding Volume Hierarchy (BVH) techniques [11]. At each level of the object representations the pairs of suspicious objects are identified by intersecting AABBs and are placed in the resulting set. The procedure begins with a pair of the given compound objects and ends when the resulting list contains only pairs of simple objects. It can be reached recursively by readdressing the original query specified for the parent objects to similar queries for their children. If the complementary octrees are not deployed, it can be done by naive intersecting AABBs of the children. If some compound objects are complemented with octrees, then it can be done more effectively using the region search query as specified above.

Because the unbalanced scenes are ubiquitous in practice and compound objects may have the extremely varied number of children, the complementary indexing structure and the described query resolution methods exhibit uniformly high performance. That is confirmed by theoretical study and practical experiments reported below.

## 4 Theoretical Study

The efficiency of indexing methods based on dynamic regular octrees was proved in the previous studies [9]. The goal of the presented investigation is to compare the proposed complementary indexing structure with its base version. It may seem that because of the deployment and updating of multiple octrees, the complementary structure is not effective and does not have a competitive advantage over the base version. However, the proved statements refute this assumption.

For brevity, we restrict ourselves to two statements, the first is concerning the complexity of the index deployment, and the second - the memory consumption for index storage. Since the worst case complexity analysis is trivial and the resulting estimations do not reflect real performance and memory consumption parameters, we provide the estimations on average using the probability theory and the introduced concepts of synthetic scenes.

**Definition 1.** *An octree with subdivisions into equal octants, strict multilevel localization of objects and upper boundary of octant cardinality  $m$  is called a regular octree and denoted as  $Octree(m)$ .*

**Definition 2.** *A set of  $n$  identical cubes with the edges of length  $0 \leq l < 1/2$  directed along the principal coordinate axes is called a synthetic one-level scene and denoted as  $S(n, l)$  if the cubes are independently and randomly distributed in a unit cube and appear successively one after another.*

**Definition 3.** *Let the synthetic multilevel scene be a dataset  $S(n, l, h)$  holding the following characteristics:*

- scene and each non-leaf object has exactly  $n \in \mathcal{N}$ ,  $n > 1$  children of equal size, uniformly distributed over the space occupied by the parent object;
- the scene tree is balanced (i.e. all the sibling objects have the same height) and the scene tree has a height  $h$ ;
- all simple objects are cubes;
- AABBs of the whole scene and its compound objects are cubes;
- assuming linear size of the parent objects is 1, the size of the child object is  $l$ ,  $0 \leq l \leq 1/2$ ;
- the scene events are successive appearances of top-level compound objects with all children.

Let us estimate the computational expenses of preparing and updating the complimentary indexing structure for modeling synthetic scenes. For this purpose, we consider a multi-level synthetic scene  $S(n, l, h)$  and compare the costs

of the base and complementary structures. To take advantage of the previous results [9], we assume that the multi-level synthetic scene is equivalent in complexity to one-level synthetic scene  $S(n^h, l^h)$  with the same number of simple objects and events. For this reason, the costs of event trees coincide and may be excluded from further comparative analysis.

It is interesting to estimate the costs of spatial indexing structures. In the mentioned work it was shown that the computational expenses required to deploy a regular octree  $Octree(m)$  for the synthetic scene  $S(n, l)$  can be estimated as follows:

$$Q = C_L n \sum_{i=1}^H \frac{(1 - 2^{i-1}l)^3}{(1 - l)^3}$$

where  $C_L$  – cost of the object localization in the child octants,  $n$  – total number of objects,  $l$  – object linear size. In this formula octree height  $H$  is defined as follows:

$$H = \begin{cases} \left\lceil \log_2 \frac{2}{l+(1-l)\sqrt[3]{\frac{m}{n}}} \right\rceil & , m \geq nl^3 \\ \lceil \log_2 \frac{1}{l} \rceil & , m < nl^3 \end{cases}$$

Then the computational expenses required to deploy the complimentary structure for the scene  $S(n, l, h)$  are estimated using the following series of expressions:

$$\begin{aligned} Q_{compactree} &= C_L n \left( \sum_{j=0}^{h-1} n^j \right) \left( \sum_{i=1}^{H_1} \frac{(1 - 2^{i-1}l)^3}{(1 - l)^3} \right) \\ &= \frac{C_L n (n^h - 1)}{n - 1} \sum_{i=1}^{H_1} \frac{(1 - 2^{i-1}l)^3}{(1 - l)^3} \\ H_1 &= \begin{cases} \left\lceil \log_2 \frac{2}{l+(1-l)\sqrt[3]{\frac{m}{n}}} \right\rceil & , m \geq nl^3 \\ \lceil \log_2 \frac{1}{l} \rceil & , m < nl^3 \end{cases} \end{aligned}$$

Here we assume that a separate regular octree is deployed for each compound object with the number of children exceeding the given threshold parameter  $m \leq n$ . No octrees are deployed if  $m > n$ . The complementary structure degenerates and becomes a scene hierarchy in this case.

**Theorem 1.** *On the assumption that computational expenses are expressed in terms of the operations of the object localization in child octants, the costs of the complimentary indexing structure and the base indexing structure for the synthetic multilevel scene  $S(n, l, h)$  are interrelated as follows:*

$$Q_{compactree} \leq \frac{(n^h - 1)}{(n^h - n^{h-1})} \frac{(1 - l^h)^3}{(1 - l)^3} Q_{octree}$$

*Proof.* Indeed, according to the mentioned above results the expenses required to deploy a dynamic regular octree for the equivalent one-level scene  $S(n^h, l^h)$  are as follows:

$$Q_{octree} = C_L n^h \sum_{i=1}^{H_2} \frac{(1 - 2^{i-1}l^h)^3}{(1 - l^h)^3}$$

$$H_2 = \begin{cases} \left\lceil \log_2 \frac{2}{l^h + (1-l^h)^{\frac{2}{3}\sqrt{\frac{m}{n^h}}}} \right\rceil & , m \geq n^h l^{3h} \\ \lceil \log_2 \frac{1}{l^h} \rceil & , m < n^h l^{3h} \end{cases}$$

Consider ratio:

$$\frac{Q_{compactree}}{Q_{octree}} = \frac{n^h - 1}{n^h - n^{h-1}} \frac{(1 - l^h)^3}{(1 - l)^3} \frac{\sum_{i=1}^{H_1} (1 - 2^{i-1}l)^3}{\sum_{i=1}^{H_2} (1 - 2^{i-1}l^h)^3}$$

Let us analyze the third factor. Since the base structure stores more objects of smaller size than any complementary octree associated with compound objects, it is obvious that  $H_1 \leq H_2$ . Also notice that  $2^{i-1}l > 2^{i-1}l^h$ , since  $0 \leq l \leq 1/2$  by definition of synthetic scenes. Therefore:

$$\frac{\sum_{i=1}^{H_1} (1 - 2^{i-1}l)^3}{\sum_{i=1}^{H_2} (1 - 2^{i-1}l^h)^3} \leq 1$$

The original ratio can be estimated as:

$$\frac{Q_{compactree}}{Q_{octree}} \leq \frac{n^h - 1}{n^h - n^{h-1}} \frac{(1 - l^h)^3}{(1 - l)^3}$$

■

It can be shown that ratio  $1 \leq \frac{n^h - 1}{n^h - n^{h-1}} \leq 2$ , given  $n \in \mathcal{N}, n > 1$ , and  $\lim_{n \rightarrow \infty} \frac{n^h - 1}{n^h - n^{h-1}} = 1$ . Also  $\frac{(1-l^h)^3}{(1-l)^3} < 8$  since  $0 \leq l \leq 1/2$  and  $\lim_{l \rightarrow 0} \frac{(1-l^h)^3}{(1-l)^3} = 1$ . Thereby, the costs of the complimentary indexing structure are limited by a constant factor compared with the costs of the base structure under the synthetic scene assumptions.

Now let us consider the memory overhead required for the complimentary indexes. For this purpose we count the number of octants in all complimentary octrees deployed and then compare it with the number of octants in the base structure deployed for the equivalent one-level scene. The overall number of the octants of the complimentary structure can be calculated as follows:

$$N_{compactree} = \sum_{i=0}^{h-1} n^i \sum_{j=1}^{H_1} 8^{j-1} = \frac{n^h - 1}{n - 1} \sum_{j=1}^{H_1} 8^{j-1} = \frac{n^h - 1}{n - 1} \frac{8^{H_1} - 1}{7}$$

while the number of octants of the base structure:

$$N_{octree} = \sum_{i=1}^{H_2} 8^{i-1} = \frac{8^{H_2} - 1}{7}$$



**Theorem 2.** *The overall number of octants in the complimentary and base indexing structures deployed for the equivalent synthetic scenes  $S(n, l, h)$  and  $S(n^h, l^h)$  are interrelated as follows:*

$$N_{compactree} = \frac{\rho^h + m}{n^h l^3} N_{octree}$$

where

$$\rho = nl^3$$

*Proof.* Consider ratio:

$$\frac{N_{compactree}}{N_{octree}} = \frac{n^h - 1}{n - 1} \frac{8^{H_1} - 1}{8^{H_2} - 1}$$

It can be shown that in a regular octree the following inequality holds:

$$\log_2 \frac{2}{l + (1-l) \sqrt[3]{\frac{m}{n}}} \leq H < \log_2 \frac{2}{l}$$

Therefore,

$$\frac{8^{H_1} - 1}{8^{H_2} - 1} \leq \frac{\frac{8}{l^3} - 1}{\left(\frac{8}{l^h + (1-l^h) \sqrt[3]{\frac{m}{n^h}}}\right)^3 - 1} \leq \frac{(8 - l^3) \left(l^h + \sqrt[3]{\frac{m}{n^h}}\right)^3}{\left(8 - \left(l^h + \sqrt[3]{\frac{m}{n^h}}\right)^3\right) l^3} \leq \frac{\rho^h + m}{n^h l^3}$$

■

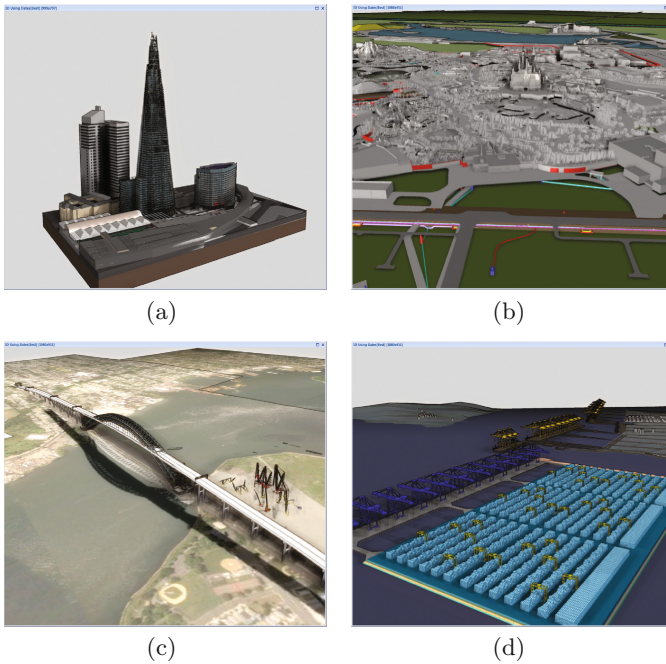
This result can be interpreted as follows. While the spatial density factor  $\rho = nl^3$  (determining the share of the total volumes of children in the volume of their parent) remains small, the number of octants in the complimentary structure is less than the number of octants in the base structure. However, if the spatial density factor grows above 1, the number of octants in the complimentary structure rises quickly in comparison with the number of octants in the base structure.

Provided statements prove that, under certain conditions, the computational and memory expenses required to deploy the complimentary indexing structure based on dynamic regular octrees are comparative with the corresponding expenses for the base structure and, therefore, will unlikely become a bottleneck in practice. Noteworthy, that the estimated computational expenses of the indexing structures reflect the complexity of the evaluation of relevant queries of the synthetic scene reconstruction, assuming that it is carried out from the beginning when no objects are placed in the scene, to the current focus time, when all objects have already appeared in the scene.

Certainly, the applied mathematical models of computations and memory consumption are rather abstract and need practical validation through a series of computational experiments with industry-meaningful spatial-temporal datasets.

## 5 Experimental Study

Let us discuss the results of the complementary index application in evaluation of queries addressed to real spatial-temporal data. Models of implementation of large-scale industrial projects presented in Fig. 4 have been selected for the tests. Parameters of the models, such as the number of simple objects that have their own geometric representation, the minimum and maximum depth of the hierarchy, the average and maximum number of children, the total number of events in a dynamic scene and the spatial occupancy of the scene are shown in Table 1, which represent a wide range of experimental data.



**Fig. 4.** Industrial models used in the computational experiments. (a) – a skyscraper construction model, (b) – a large-grained park development plan, (c) – a bridge construction plan, (d) – a construction and equipping plan for a port warehouse.

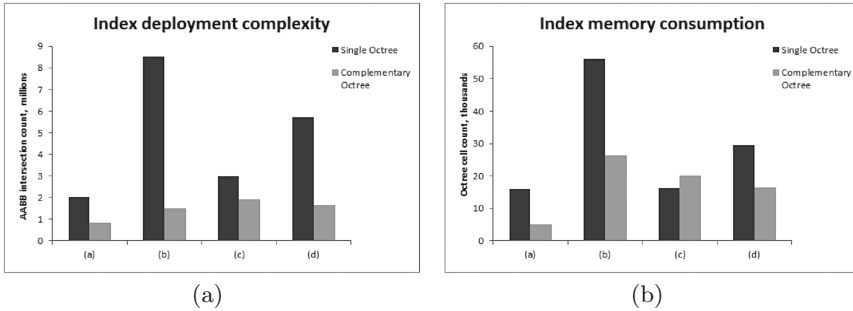
The models chosen for testing contain tens or hundreds of thousands of geometric objects that allows to consider them as large-scale data. All the objects are organized in hierarchies, but their structures are very unbalanced.

Model (a) is a detailed spatial-temporal model describing the progress of construction works for a skyscraper in accordance with an approved schedule. This model is characterized by the presence of many small parts, each of them has its own dynamic behavior. Model (b) describes a large-grained park development plan consisting of several parts where the details of their implementation are not

**Table 1.** Parameters of the models chosen for testing.

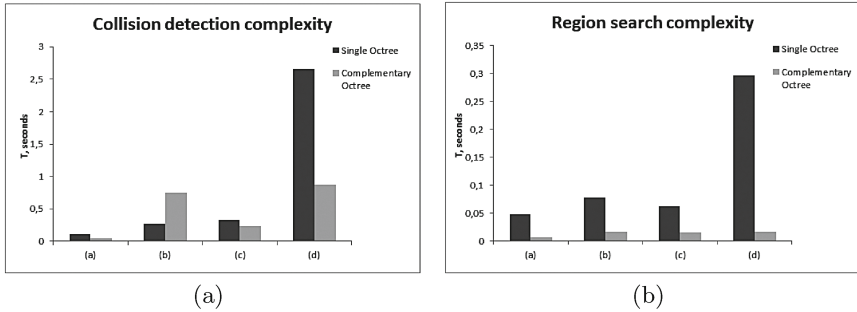
	Model (a)	Model (b)	Model (c)	Model (d)
Number of simple objects	73731	188465	85776	169606
Number of events	6877	6105	4381	752
Maximum hierarchy height	13	25	18	8
Minumum hierarchy height	5	3	2	3
Maximum number of children	5526	16625	21816	31932
Average number of children	4.5286	1.66046	5.79458	1.99634
Spatial occupancy of the scene	0.505344	0.140742	0.0108893	0.691711

specified. Visually, realization of each part of the plan leads to the simultaneous appearance of an independent model consisting of a large number of objects in the scene. Model (c) implements a construction plan for a bridge taking into account the deployment and use of construction equipment. A feature of the model is the extension of individual elements of the construction. Finally, model (d) describes a construction and equipping plan for a warehouse at a port. The peculiarity of the model consists in the development of logistics scenarios implying the movement of large amounts of cargo and port facilities.



**Fig. 5.** Comparison of deploying cost and memory expenses using octree and complementary index for test models (a), (b), (c), (d)

The experiments have been conducted using a typical desktop computer: Core i7 3770 3.4GHz, 16 GB RAM (1600MHz). First of all, compare the cost of deploying the spatial indexes for these test models. The comparison results are shown in Fig. 5 and illustrate the fact that the computational cost and the memory expenses required for the construction of the complementary index do not exceed and often lower the cost of deploying a regular octree for the equivalent model containing only simple objects of the scene. Thus, the construction of the complementary index does not lead to additional overhead as compared with traditional structures and confirms the theoretical conclusions.



**Fig. 6.** Comparison of time required for collision detection and region search using octree and complementary index for test models (a), (b), (c), (d)

Let us analyze the time required to evaluate typical collision detection and region lookup queries in a scene. Figure 6 shows the results of comparative testing of the complementary and basic indexing structures. Note that the detailed search of intersections, the efficiency of which is mainly determined by using special methods for intersection of curves, surfaces, solids, was not implemented. Instead, their preliminary localization using AABB has been performed.

It can be seen that in most cases the computational cost for collision detection using the complementary indexing structure is much lower than with regular octree. This can be explained by the fact that object hierarchies in the industrial models reflect not only a logical structure of assembling the components, but also their natural spatial composition. As a result, collisions between compound objects appear infrequently. This allows you to make decisions about potential intersections of objects already at the higher levels of the hierarchy and avoid traversing and analysis of simple objects. However, if a hierarchical structure does not reflect the spatial object composition (as it happens in the model (b)) the complementary index may lose performance.

In the experiments on region search by area of size equal to  $\frac{1}{2}$  a stage for each dimension was generated 100 times, the position was selected randomly, and then the results were summarized. The results are shown in Fig. 6(b). The experiments demonstrate that the computational complexity of the region search in the case of using the base structure greatly exceeds the cost of using the complementary index. This fact can be explained by the possibility to return a verdict of belonging a compound object to the given spatial region without extensive analysis of simple objects.

## 6 Conclusions

We have presented the spatial-temporal data model as a scene that admits hierarchically organized, spatially heterogeneous objects with individual temporal behaviors. The data model is quite general, allowing its use in different science

and industry applications. For the model presented we have argued the relevance of dynamic event-driven regular octrees as an underlying spatial-temporal indexing structure and have shown its effectiveness for queries such as scene reconstruction, region search, and collision detection. For hierarchically organized scenes the complementary generalization of the indexing structures has been proposed. Its advantages over traditional structures have been confirmed by theoretical conclusions and practical experiments with industry meaningful datasets originated from the construction modeling applications.

## References

1. Carvalho, A., Ribeiro, C., Augusto Sousa, A.: A spatio-temporal database system based on timeDB and oracle spatial. In: Tjoa, A.M., Xu, L., Chaudhry, S.S. (eds.) CONFENIS 2006. IIFIP, vol. 205, pp. 11–20. Springer, Boston, MA (2006). [https://doi.org/10.1007/0-387-34456-X\\_2](https://doi.org/10.1007/0-387-34456-X_2)
2. Griffiths, T., et al.: TRIPOD: a spatio-historical object database system. In: Ladner, R., Shaw, K., Abdelguerfi, M. (eds.) Mining Spatio-Temporal Information Systems. The Kluwer International Series in Engineering and Computer Science, vol. 699. Springer, Boston (2002)
3. Oracle Corporation: Oracle Spatial User's Guide and Reference, 10g Release 1 (10.1) (2003)
4. Semenov, V.A., Kazakov, K.A., Zolotov, V.A.: Global path planning in 4D environments using topological mapping. In: Gudnason, G., Scherere, R. (eds.) eWork and eBusiness in Architecture, Engineering and Construction, pp. 263–269. CRC Press, Taylor & Francis Group, London, UK (2012)
5. Nandal, R.: Spatio-temporal database and its models: a review. IOSR J. Comput. Eng. **11**(2), 91–100 (2013)
6. Seo-Young, N.: Literature Review on Temporal, Spatial, and Spatiotemporal Data Models: Computer Science Technical Reports. Paper 150 (2004). [http://lib.dr.iastate.edu/cs\\_techreports/150](http://lib.dr.iastate.edu/cs_techreports/150)
7. Nguyen-Dinh, L.-V., Aref, W.G., Mokbel, M.F.: Spatio-temporal access methods: Part 2. IEEE Data Eng. Bull. **33**(2), 46–55 (2010)
8. Menninghaus, M., Breunig, M., Pulvermuller, E.: High-Dimensional Spatio-Temporal Indexing. Open J. Databases **3**(1), 1–20 (2016)
9. Zolotov, V.A., Petrishchev, K.S., Semenov, V.A.: Methods of spatial indexing of dynamic scenes based on regular octrees. Program. Comput. Softw. **42**(6), 375–381 (2016)
10. Semenov, V.A., Anichkin, A.S., Morozov, S.V., Tarlapan, O.A., Zolotov, V.A.: Visual planning and scheduling of industrial projects with spatial factors. In: Bil, C., Mo, J., Stjepandic, J. (eds.) Proceedings of 20th ISPE International Conference on Concurrent Engineering. IOS Press, Melbourne, Australia, pp. 343–352 (2013). ISBN: 978-1-61499-301-8
11. Dinas, S., Bañón, J.M.: A literature review of bounding volumes hierarchy focused on collision detection. Ingeniería Y Competitividad **17**(1), 49–62 (2015)

# An Approach to the Validation of XML Documents Based on the Model Driven Architecture and the Object Constraint Language

Denis A. Nikiforov<sup>(✉)</sup>, Dmitriy V. Korj, and Ruslan L. Sivakov

Center of Information Technologies LLC, Ekaterinburg, Russia  
{Denis.Nikiforov,Dmitriy.Korj,Ruslan.Sivakov}@centre-it.com

**Abstract.** It is possible to develop data processing applications using a variety of different data representation formats (EDI, CSV, XML, JSON), domain-specific languages, and general-purpose programming languages (XSLT, SQL, Java, C#). On the one hand, such a variety allows one to choose the most optimal data format or language based on the specific requirements being applied, while on the other one, contemporary information systems or complexes of integrated information systems have become similar to the Tower of Babel, being so cumbersome to build and maintain. A possible solution to this issue could be found in developing platform-independent specifications to be used for generating the source code for each required platform.

This article describes an approach to the XML document validators' generation based on UML models with Object Constraint Language (OCL) rules. The authors give a brief account of similar tools and propose a generalized schema for generating the validators based on a model-driven approach. The core component of this schema is the transformation of OCL constraints to XPath assertions. The first ones could come from one of the supported platform-independent models (Eurasian Economic Union Data Model or ISO 20022), while the later could be embedded into XML Schema 1.1, XSLT or Java code.

The transformation is implemented at the model level in the Query/View/Transformation language. The article does not go into details of converting OCL into XPath, because such a description takes up a lot of space and has already been given in similar articles. The authors describe only the key features of their approach: development of metamodels for XPath, XSD 1.1, and XSLT, support of a variety of platform-independent and platform-specific models, determination of elements subject to validation, external data sources, kinds of validation messages, preconditions.

**Keywords:** XML validation · Semantic validation  
Unified Modeling Language · Object Constraint Language  
XPath · XML Schema · XSLT · Model Driven Architecture  
Platform-independent model · Metamodel · Model transformation  
Query/View/Transformation · Eclipse Modeling Framework  
ISO 20022

# 1 Introduction

Consider the following example. Let us say you are designing a website where users can post their resumes (Fig. 1). Among other things, users are to specify an employment period for each job they had, however they only specify a starting date for their currently held job. You could introduce the following constraint to ensure consistency of each job’s dates: if an end date is specified, then it must not be earlier than the starting date. Such a restriction can be easily implemented in JavaScript.

**Fig. 1.** Example of a data-entry form for resume details

Subsequently, you decide to make it easier for users to add their resumes to your site, and implement an option allowing them to upload the resumes in an XML format:

```
<CurriculumVitae>
  <!-- ... -->
  <WorkExperience><!-- Past Employment -->
    <StartDate>2015-01-01</StartDate>
    <EndDate>2017-03-31</EndDate>
    <Organization>Company A Inc.</Organization>
  </WorkExperience>
  <WorkExperience><!-- Current Employment -->
    <StartDate>2017-04-01</StartDate>
    <Organization>Company B Inc.</Organization>
  </WorkExperience>
</CurriculumVitae>
```

You can use an existing JavaScript code to validate the XML documents if the server-side logic is implemented on Node.js, however this is not always the case, and you may need to do a repeat implementation of the constraints by means of XSD 1.1, XSLT, or a general-purpose programming language, such as Java.

Let us assume that despite the implemented constraints your application's database still receives inconsistent data. In order to ensure the data's integrity, you define the constraints in a third language – SQL. This duplication may result in a disagreement of implementations and an increase in the application's development and maintenance costs.

The Object Management Group (OMG) proposes to use the Model Driven Architecture (MDA) to solve this problem [15]. Towards this end, you would define the data schema and constraints in a Platform-Independent Model (PIM), e.g. in UML [28] and OCL [19,26] languages, correspondingly. The constraint under consideration could be defined in OCL in the following manner:

```
WorkExperience->forall(x |
  x.EndDate->notEmpty() implies x.StartDate <= x.EndDate)
```

You transform the PIM into various Platform-Specific Models (PSM) whenever the need to implement constraints for a particular platform arises, those PSMs being used as a basis for generating the required source code [1,5,16].

This article discusses only the transformations from PIM to PSM, used to generate validators for XML documents. Implementation of constraints for web forms or relational databases lies outside this article's scope.

The paper is organized as follows. In Sect. 2, we define and compare the existing XML document validation tools using OCL constraints. In Sect. 3, we propose a generalized framework for generating validators based on the MDA. In Sect. 4, we define some special aspects of our approach that distinguish it from its analogues. In Sect. 5, we list areas to focus on in the future.

## 2 Overview of Analogues

There are two approaches to validating XML documents using OCL rules: (1) transforming the rules into expressions in a language that can be run on a certain technology platform (Schematron [10], XPath, Java, etc.) or (2) interpreting the rules within the context of an XML document's object model [25].

In the first case, the OCL rules get adapted to run on a desired technology platform (Fig. 2), while in the second case it is the data that gets adapted from its platform-specific representation to a suitable platform-independent form (Fig. 3).

The first approach is preferred in situations where the environment, in which the XML documents are to be validated, has to comply with stringent requirements. However, if the part played by overhead costs associated with generating an object model of the XML document is insignificant, and there are no restrictions on the choice of technologies, it is more appropriate to use the second approach. This is due to the fact that generating an object model for data is much simpler than mapping all the operations defined in the OCL Standard Library.

Table 1 describes the existing tools for XML documents validation using OCL rules. The OCL specification's support provided by tools based on the OCL interpreter is more complete. However, the most recent versions of OCLE and



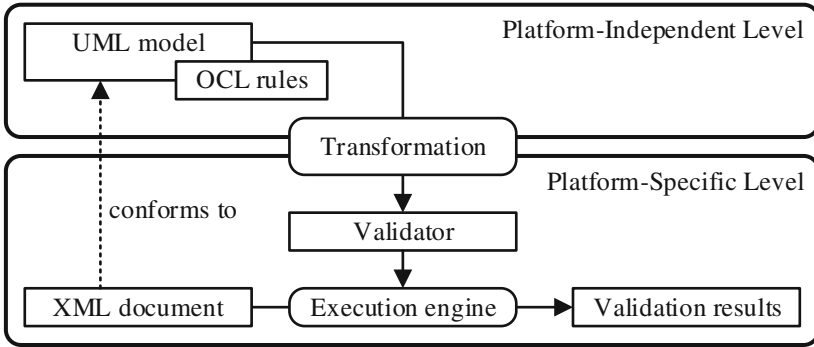


Fig. 2. Flowchart of transformation of OCL constraints

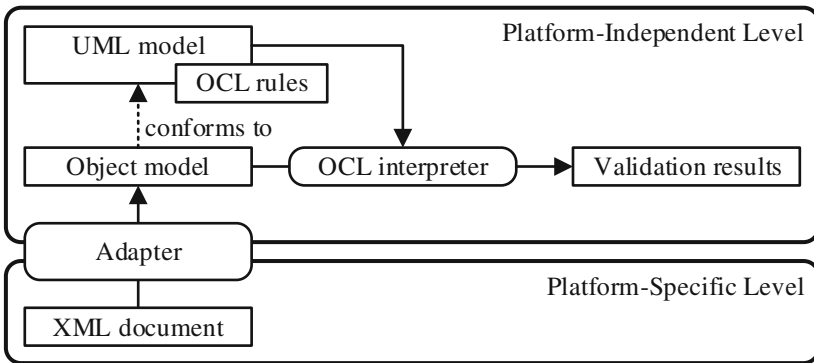


Fig. 3. Flowchart of interpretation of OCL constraints

Dresden OCL date back, respectively, to years 2005 and 2015, meaning that they implement older versions of the specification. Unlike other Schematron-generating tools, eXolutio supports relatively complex “iterate” and “closure” operations, however it is not clear if “substring” is supported [13].

eXolutio and OCLE support plain UML class diagrams as a PIM. ShapeChange and NIEM transformation support UML models as per ISO 19109 and NIEM specification, respectively. For Eclipse OCL, either a simple UML class diagram or an Ecore model adapted for the Pivot Meta-Model can be used as a PIM. Dresden OCL additionally provides adapters for Java classes and XML schemas.

Some tools allow to extend the OCL Standard Library by using custom UML classes with OCL operations. The latter are either interpreted, or transformed to XSLT, XQuery or Java functions.

External data sources are fully supported only by OCLE. All data sources are defined in a single logical model, and can be referenced from the OCL rules.

For eXolutio, an OCL syntax extension to reference external XML documents is proposed in [13].

Most tools are implemented in Java or other general-purpose programming languages. The only exclusion to this is the NIEM PIM to Schematron transformation, having been implemented in QVTo [30], however even in its case the XPath expressions are generated directly as text, bypassing PSM.

**Table 1.** Comparison of analogues

Tool	Validation mechanism	OCL support	Source PIM	OCL extension mechanism	Support of external sources	Implementation
Shape-Change <sup>a</sup>	Schematron	Extremely limited	ISO 19109	Absent	Absent	Java
NIEM [27]	Schematron	Limited	NIEM	Absent	Absent	QVTo
eXolutio [11, 14]	Schematron	Average	UML class diagram	Present	Proposed	Unknown
Dresden OCL [3]	Interpretation, Java	Potentially complete	Adapted model	Present	Absent	Java, EMF
Eclipse OCL [31]	Interpretation	Complete	Adapted model	Present	Absent	Java, EMF
OCLE [2]	Interpretation, Java	Potentially complete	UML class diagram	Present	Present	Unknown

<sup>a</sup><http://shapechange.net/targets/xsd/extensions/ocl/>.

### 3 Proposed Generalized Validator Generation Scheme

Most of the above-discussed tools are solid applications written in general-purpose programming languages. This significantly complicates understanding of how exactly the UML models and OCL rules are transformed in those tools, and makes it difficult to adjust transformations to support other PIM or to better support the OCL Standard Library.

MDA offers an alternative approach to developing applications, where all source, intermediate and target artifacts are considered to be models within a certain modeling space [4]. A development process can be represented as a sequence of model transformations [15]. Each model must conform to a certain metamodel, and each metamodel should be developed in accordance with a certain meta-metamodel. Each meta-metamodel forms its own modeling space [4].

OMG has developed a number of specifications describing metamodels (UML, OCL), meta-metamodels (MOF), model transformation languages (QVTo), text representation of models (XMI). Eclipse Modeling Project (EMP) [6] implements some of these specifications. There are various specifications and tools for model-based development, however, OMG is the main standardizing organization in this field and offers the most complete set of specifications, while the most complete implementation of the specifications is provided by the EMP.

Figure 4 shows our proposed and implemented generalized scheme of validator generation based on UML models with OCL constraints. Metamodels and transformations developed by us at [21–23] are marked in gray in the figure. All

text artifacts belong to the EBNF modeling space. UML models, XSLT models, OCL Abstract Syntax Trees (AST), XPath AST belong to the Ecore [31] modeling space (Ecore is an analog of MOF in EMP). The figure depicts the sequence of transformations for a single pair of PIM and PSM. It is possible to use not only XSLT, but also XSD 1.1 or Java as a PSM.

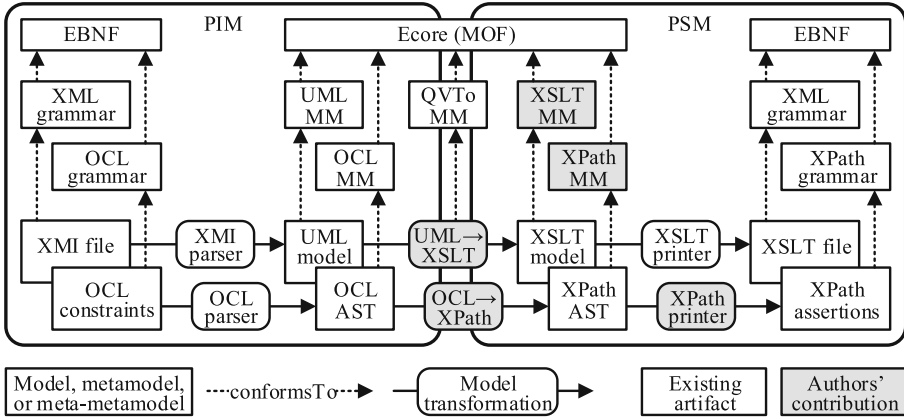


Fig. 4. Proposed generalized validator generation scheme

The transformation is done in three stages. First, the UML model with OCL constraints is translated from a text representation in an EBNF modeling space to an Ecore modeling space [4]. Subsequently, the Ecore modeling space is used to transform PIM to PSM, with the transformation being defined in QVTo. Finally, the PSMs are converted to text and sent back to the EBNF modeling space. Let us review each step in more detail.

### 3.1 Transformation of PIM from Textual to Ecore Representation

The source UML model is represented in the XMI format [29]. The XMI file contains OCL rules. We use Eclipse UML and Eclipse OCL to parse them [6]. The result is an UML model conforming to an UML metamodel and a set of OCL models (abstract syntax trees) conforming to an OCL metamodel. Both metamodels are based on the Ecore meta-metamodel, so their subsequent transformations can be described in the QVTo language.

### 3.2 Transformation of PIM to PSM

Currently, two kinds of source PIMs (data model of the Eurasian Economic Union (EAEU)<sup>1</sup> and ISO 20022 [9, 12, 24]) and three kinds of target PSMs (XML

<sup>1</sup> <https://eomi.eaeunion.org>.

Schema, XSLT, and Java) are supported by the transformation. A separate QVTo transformation is implemented for each of the six PIM and PSM combinations, while the main part – which is the transformation of OCL to XPath – is reused. In order to implement these transformations via Eclipse QVTo it is necessary to define metamodels based on the Ecore meta-metamodel for each source and target model. For UML and OCL the metamodels are implemented within the EMP.

Moreover, this project includes an implementation of the XSD 1.0 meta-model. We cannot use it in our generator, because it is impossible to embed XPath assertions in XSD 1.0. An XSD 1.1 metamodel is required. The later itself is an XML-based language, it is defined in the corresponding XML schema. We imported this schema with the wizard provided by the Eclipse Modeling Framework (EMF) [17,31], and created an Ecore-based metamodel for XSD 1.1 [21]. Similarly, we have generated on the basis of the XML schema an Ecore-based metamodel for XSLT 2.0 [21].

In addition to XSD and XSLT, our transformation also generates Java code and XPath expressions that are not XML-based languages and that has not XML schemas. Their syntax is defined in EBNF, making it necessary to use other tools to develop Ecore-based metamodels. One of such tools is EMFText [8, 18]. It allows to define a textual representation in an EBNF-like language for Ecore-based metamodels. For Java, an Ecore-based metamodel has already been implemented in the JaMoPP project [7]. For XPath 2.0, we have implemented our own metamodel and described its syntax [22]. General principles of design of Ecore-based metamodels based on EBNF grammars were described by us in [20] using the example of SQL.

### 3.3 Transformation of PSM from an Ecore Representation to a Textual One

Serialization of Ecore models for XML-based languages (XSD, XSLT) is a trivial task. When importing an Ecore-based metamodel from an XML schema the former is complemented by extended metadata that allow to (de-)serialize Ecore models not just in XMI format, but also in the form of XML documents that conform to the original XML schema. It is possible to generate parsers and printers automatically for languages implemented by means of EMFText. For Java, they are available in the JaMoPP project, for XPath 2.0 – in [22].

### 3.4 Summary

In a general case scenario, in order to transform  $n$  PIMs to  $m$  PSMs it is necessary to develop  $n \times m$  transformations. In this case, such transformations number  $2 \times 3 = 6$ . This nonlinear relationship complicates the development of validator generators. The difference between our approach and the approaches described in Sect. 2 is that we have decomposed the transformations as much as possible.

Transformations from a text representation (Sect. 3.1) or into a text representation (Sect. 3.3) are generalized and reusable. Their number is linearly dependent on the number of PIMs and PSMs to be supported by the generator of validators. Some of these transformations had already been implemented (Eclipse XMI parser), other ones we implemented on our own (XPath 2.0 printer) [22].

The transformation from OCL to XPath 2.0 is also generic and does not depend on the PIM and PSM combination. This is described in further details in Sects. 4.1 and 4.2.

It is these transformations that are the most difficult to implement, whereas transformations that are specific to each pair of PIM and PSM are more numerous, but easier to implement.

## 4 Features Distinguishing Our Approach from Its Analogues

As seen in the case of tools discussed in Sect. 2, the rules of OCL transformation to XPath generally coincide. Therefore, we will not describe them in detail in this paper, especially since the OCL Standard Library defines around two hundred operations. The description of the rules is quite voluminous. Hence, we will describe only those features of our approach that distinguish it from its peers in a fundamental way.

### 4.1 Different Source PIMs

Each of the analogues described in Sect. 2 can generate validators only for certain kinds of PIM. If you used a different PIM, you would not be able to use any of these tools, or you would have to adapt your PIM somehow. Our generator of validators supports two kinds of PIM (EAEU data model and ISO 20022) and can be easily enhanced to support additional kinds of models.

This is achieved as follows. A significant part of OCL expressions is converted to XPath expressions without regard to which data model they are used in. For example, the trivial “ $2 + 2 <> 5$ ” expression of OCL is always converted to a “ $2 + 2 \text{ ne } 5$ ” XPath expression, regardless of the applicable PIM.

The specifics of various types of PIMs appear only when converting expressions that include calls to object properties (PropertyCallExp). Different PIMs use different UML stereotypes for data elements, data types, attributes, and external data sources. Moreover, their modeling can be done using different UML elements. For example, the data model of EAEU models reusable data elements as classes, while ISO 20022 allows only local elements that are modeled as properties of classes. In order to put aside these differences, the following abstract operations are declared in our transformation: “isDataType”, “isDataElement”, “isAttribute”, “isExternalSource”. These operations have to be implemented to add support for a new PIM.

Moreover, each PIM requires “getQName” and “getUnprefixedQName” operations, that return the model object’s name with and without the namespace

prefix, correspondingly, to be implemented. Finally, it is necessary to implement the following operations that allow to abstract the OCL transformation to XPath from primitive type systems which tend to vary from one PIM to another: “isNumericType”, “isStringType”, “isBooleanType”, “isDateType”, “isDateTimeType”, “isTimeType”, “isDurationType”.

The tools discussed in the Sect. 2 have their own private OCL to XPath transformations. Our experience shows that it is possible to develop a generalized transformation treated separately from the original PIM. It is also possible to develop a generalized system of primitive types that unifies the type systems of different PIMs.

## 4.2 Different Target XPath Host Languages

OCL expressions are meaningful only for some UML or Ecore (MOF) models. Similarly, XPath expression cannot be interpreted by themselves, they are to be embedded into some host language. Our tool supports three host languages: XSD 1.1, XSLT, and Java.

In contrast to the source PIM, the target PSM (host language) does not affect the OCL transformation rules to XPath. Only the upper levels of ASTs of OCL expressions have different ways of transformation. If they contain any “forAll” or “implies” operations, those get transformed to host language expressions, not to XPath expressions. Further details can be found in Sects. 4.4, 4.5, and 4.6.

Table 2 lists the features of XPath host languages that our generator of validators supports. XSD, unlike XSLT and Java, serves for basic structural validation of XML documents. It is much more difficult to check the sequence of XML elements or to make sure there are no unexpected elements through XSLT or Java. Therefore, it is necessary to combine validators based on such technologies with XML schema-based checks.

XML schema 1.1, as opposed to its version 1.0, may be supplemented with XPath assertions that implement relatively complex validation rules. However, in XML schemas the rules are defined for data types, while in the case of using XSLT or Java the XPath processor usually will not be datatype-aware. Hence, if an XML document reuses the same data type for several data elements, one has to duplicate constraints for each element. This is not always a problem, in business applications it is often the elements that are semantically constrained, not the data types, so it is easier to implement the rules using XSLT or Java than XSD.

Many information systems already use XML schemas for validation of XML documents. If XPath assertions generated from the OCL constraints complement these, no further changes to the validator are necessary. Using XSLT or, especially, Java can require significant changes in the execution environment.

However, when using the XML schema, you have almost no influence on the way validation results are presented. XSLT and Java allow to generate different types of messages in the required representation. See more details in Sect. 4.6.

Finally, XML schemas do not allow to validate XML documents using external data source or remote services. For example, they cannot check a code against

**Table 2.** Outline of host languages

Feature	XSD 1.1	XSLT	Java
Basic structural validation	Yes	No	No
Rules are specified for	Data types	Data elements and data types <sup>a</sup>	Data elements
Execution environment requirements	Low	Moderate	High
Customization of validation results representation	No	Yes	Yes
External data sources and complex constraints	No	Maybe (via extensibility)	Yes

<sup>a</sup>Only for schema-aware XSLT processors.

a code list stored in the database. XSLT (via extensibility) and Java do have the capability to use external data sources.

### 4.3 External Data Sources

Sometimes you need to check the data contained in an XML document using remote services or external data sources. For example, code values might need to be checked against code lists stored in relational databases. The currently existing tools offer only a very limited support for such data validation rules. Our tool supports two kinds of checks – one with remote and another with local execution of expressions. The first kind corresponds to OCL expressions that include the “allInstances” operation. For example, in the following constraint the database is checked for any resumes with the same telephone number but a different email address:

```
not CurriculumVitae.allInstances()->exists(x |
    x.Email <> self.Email and x.Phone = self.Phone)
```

The body of “exists” iteration is transformed to an XPath expression for its remote execution:

```
fn:not(ext:exists("CurriculumVitae", fn:concat(
    "Email != ", Email, " and Phone = ", Phone)))
```

For simplicity reasons, we do not consider the quoting of parameter values. Many databases support such XPath queries over XML documents.

For the second kind of rules there should be an external data source or service defined in the UML model, e.g. “ExternalService1”:

```
ExternalService1::isValidPhone(Address/Country, Phone)
```

A procedure is defined for the service that validates phone numbers against the country indicated in the address. In this case all XPath expressions are computed locally, and only the results are passed to the remote service:

```
ext:call("ExternalService1", "isValidPhone",
        Address/Country, Phone)
```

Validation of a code against a code list can be implemented in either way. In the first case the UML model should define the code list's structure, while in the second it is the interface to access it that has to be defined. Note that XPath and OCL are not intended to describe operations with complex logic. In principle, some constraints can be implemented in these languages, but if such constraints are too complicated, it is advisable to implement them through external services.

#### 4.4 Determination of XML Elements Subject to Validation

OCL constraints are always tied to a specific UML classifier. When checking an OCL constraint, instances of this classifier are considered to be valid or invalid. Accordingly, when converting an OCL constraint to an XPath assertion, fragments of XML documents containing data on the instance are considered valid or invalid.

The validation rules are often specified for the entire message, not for individual data elements. An example of such a rule is the OCL constraint specified in Sect. 1. The following XPath expression would be generated for it:

```
every $x in WorkExperience satisfies
  fn:not($x/EndDate) or $x/StartDate <= $x/EndDate
```

If there is at least one entry in the work experience with the starting date falling later than the end date, an error will be issued for the entire resume. XPath, as well as OCL, is an expression language, and it lacks the statements to display error messages. In order to display error messages job-by-job one must expand the “forAll” iteration in the following way:

```
<xsl:for-each select="WorkExperience">
  <xsl:choose>
    <xsl:when test="fn:not(EndDate) or StartDate <= EndDate">
      <!-- Success --></xsl:when>
    <xsl:otherwise><!-- Error --></xsl:otherwise>
  </xsl:choose>
</xsl:for-each>
```

The current context item is the current validation element. For Java or another host language, the “forAll” iteration is expanded in a similar way.

Surely, this will give us more informative validation results, however the area of validation can be narrowed even further. To do this you must find in the OCL rule all sub-expressions that returns values of some data elements. In our case



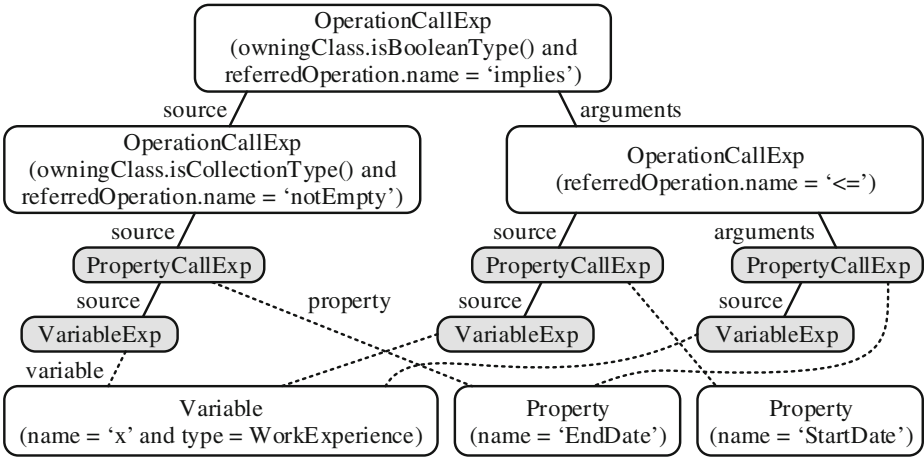


Fig. 5. Example of an abstract syntax tree of an OCL expression

those are the sub-expressions marked gray on Fig. 5: “StartDate” and “EndDate”. In general, it can be not only “PropertyCallExp”, but “IteratorExp”, as well.

### 4.5 Preconditions

In our example the current job (with a missing “EndDate”) would result in a message declaring that the validation was successful, which is not quite proper, because “StartDate” and “EndDate” were not compared with each other. Often it is practical to distinguish between the following situations: 1) the precondition was met and the test completed successfully; and 2) the precondition was not met and the test was not completed. In such a case, OCL expressions containing the “implies” operation can be expanded as follows:

```
<xsl:choose>
  <xsl:when test="EndData">
    <xsl:choose>
      <xsl:when test="StartDate <= EndDate">
        <!-- Success --></xsl:when>
      <xsl:otherwise><!-- Error --></xsl:otherwise>
    </xsl:choose>
  </xsl:when>
  <xsl:otherwise><!-- Not checked --></xsl:otherwise>
</xsl:choose>
```

### 4.6 Kinds of Validation Messages

Usually XML document validation tools allow to specify for each rule the error message that the user sees in case of a data element failing the check. We have

implemented five kinds of messages in our tool: success, error, not checked (see Sect. 4.5), not found (see Sect. 4.4), not supported (see Sect. 4.3).

```
<xsl:variable name="__items" as="item()*">
  <xsl:for-each select="WorkExperience">
    <xsl:choose>
      <xsl:when test="EndData">
        <xsl:choose>
          <xsl:when test="StartDate <= EndDate">
            <!-- Success --></xsl:when>
          <xsl:otherwise><!-- Error --></xsl:otherwise>
        </xsl:choose>
      </xsl:when>
      <xsl:otherwise><!-- Not checked --></xsl:otherwise>
    </xsl:choose>
  </xsl:for-each>
</xsl:variable>
<xsl:copy-of select="$__items" />
<xsl:if test="fn:empty($__items)"><!-- Not found --></xsl:if>
```

Messages may contain embedded OCL expressions:

`{{StartDate}}` must not be greater than `{{EndDate}}`

In general, the embedded expressions can be more sophisticated. Their abstract syntax tree can be analyzed as described in Sect. 4.4, and data elements that served as basis for the computed value can be determined.

## 5 Conclusion

In Sect. 1 we gave an example of a platform-independent formalization of a constraint in OCL. Section 4.6 presents a fragment of an XSLT code that can be generated based on this constraint using our approach and tools [21–23].

Generating XML document validators based on UML models with OCL constraints cannot be reduced to the transformation of OCL rules into XPath assertions. We have shown in this article that by analyzing the abstract syntax trees of OCL expressions it is possible to generate various subprograms in different host languages (XSLT, Java, etc.) that would test the preconditions, determine elements subject to validation, and output different kinds of validation messages.

Currently, our generator of validators supports only one-third of the OCL Standard Library. This is sufficient for its use in commercial projects. For example, the “closure” iteration that is not supported for the time being, in effect sees very little use in semantic business rules. Subsequently, we plan to implement a full support for the OCL and describe in detail its transformation to XPath.

Our experience shows that OCL allows to specify constraints that are syntactically correct, but make no sense from a business point of view. For example, the rules should be free of preconditions that always return false values or always

true values, they should not contain checks for presence of required elements in a document, etc. A number of similar semantic checks of OCL expressions must be performed by the generator. We plan to describe them in more detail.

Further, a formal proof of the validity of OCL to XPath transformation is an important task. None of the existing generators guarantee that the resulting XPath assertions are semantically equivalent to the original OCL constraints.

Finally, another problem pertaining to the development of generators is the lack of a truly platform-independent system of primitive data types. For example, primitive types of ISO 20022 effectively duplicate the XSD data types. That is why the data types are mapped one to one when converting OCL constraints to XPath assertions. On other platforms (SQL, Java, etc.) data types may differ significantly from XSD or ISO 20022, making it difficult to generate code for these platforms. In fact, ISO 20022 is tied to a single platform – XML.

## References

1. Cabot, J., Teniente, E.: Constraint support in MDA tools: a survey. In: Rensink, A., Warmer, J. (eds.) *ECMDA-FA 2006*. LNCS, vol. 4066, pp. 256–267. Springer, Heidelberg (2006). [https://doi.org/10.1007/11787044\\_20](https://doi.org/10.1007/11787044_20)
2. Chiorean, D., Bortes, M., Corutiu, D.: Object constraint language environment, a tool supporting teaching and learning UML and OCL, the understanding and using of metamodeling, abstraction and design by contract. In: *Eight Workshop on Pedagogies and Tools for Teaching and Learning Object Oriented Concepts (2000)*
3. Demuth, B., Hussmann, H., Loecher, S.: OCL as a specification language for business rules in database applications. In: Gogolla, M., Kobryn, C. (eds.) *UML 2001*. LNCS, vol. 2185, pp. 104–117. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45441-1\\_9](https://doi.org/10.1007/3-540-45441-1_9)
4. Djurić, D., Gašević, D., Devedžić, V.: The tao of modeling spaces. *J. Object Technol.* **5**, 125–147 (2006)
5. Gaafar, A., Sakr, S.: Towards a framework for mapping between UML/OCL and XML/XQuery. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) *UML 2004*. LNCS, vol. 3273, pp. 241–259. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30187-5\\_18](https://doi.org/10.1007/978-3-540-30187-5_18)
6. Gronback, R.C.: *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, Boston (2009)
7. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Construct to reconstruct - reverse engineering Java code with JaMoPP. In: *Reverse Engineering Models from Software Artifacts - REM 2009 (2009)*
8. Henriksson, J., Heidenreich, F., Johannes, J., Zschaler, S., Abmann, U.: Extending grammars and metamodels for reuse: the Reuseware approach. *IET Softw.* **2**(3), 165–184 (2008)
9. ISO: *Financial services - Universal financial industry message scheme - Part 1: Metamodel*. ISO 20022-1, International Organization for Standardization (2003)
10. ISO: *Information technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron*. ISO/IEC 19757-3:2016, International Organization for Standardization (2016)
11. Klmek, J., Malý, J., Necaský, M., Holubová, I.: eXolutio: methodology for design and evolution of XML schemas using conceptual modeling. *Inform. Lith. Acad. Sci.* **26**, 453–472 (2015)

12. Korchagin, A.B., Lisikh, I.G., Nikiforov, D.A., Sivakov, R.L.: Data models for information exchange. *Int. J. Open Inf. Technol.* **5**(3), 49–55 (2017)
13. Malý, J.: XML Document Adaptation and Integrity Constraints in XML. Ph.D. thesis, Charles University in Prague (2013)
14. Malý, J., Nečaský, M.: Evaluation of OCL expressions over XML data model. *J. Univ. Comput. Sci.* **20**, 329–365 (2014)
15. Miller, J., Mukerji, J.: MDA guide version 1.0.1 (2003). <http://www.omg.org/cgi-bin/doc?omg/03-06-01>
16. Moskal, J., Kokar, M., Morgan, J.: Semantic validation of T&E XML data. In: *International Telemetering Conference Proceedings*, International Foundation for Telemetering, October 2015
17. Nikiforov, D.A.: Development of metamodels using the Eclipse Modeling Framework, September 2015. <https://habrahabr.ru/company/cit/blog/266433/>
18. Nikiforov, D.A.: An introduction to the development of DSLs using EMFText, November 2015. <https://habrahabr.ru/company/cit/blog/270483/>
19. Nikiforov, D.A.: The Object Constraint Language, August 2015. <https://habrahabr.ru/company/cit/blog/264963/>
20. Nikiforov, D.A.: Development of the parser, printer, and editor for SQL using EMFText, December 2016. <https://habrahabr.ru/company/cit/blog/271945/>
21. Nikiforov, D.A.: Ecore-based metamodels of XML Schema 1.1 and XSLT 2.0, February 2017. <https://doi.org/10.5281/zenodo.291483>
22. Nikiforov, D.A.: The EMFText-based metamodel, parser, and printer of XPath 2.0, February 2017. <https://doi.org/10.5281/zenodo.291481>
23. Nikiforov, D.A.: UML to XML Schema 1.1 transformation, version 1.1, February 2017. <https://doi.org/10.5281/zenodo.291482>
24. Nikiforov, D.A., Korchagin, A.B., Sivakov, R.L.: An ontology-driven approach to electronic document structure design. In: Ignatov, D.I., Khachay, M.Y., Labunets, V.G., Loukachevitch, N., Nikolenko, S.I., Panchenko, A., Savchenko, A.V., Vorontsov, K. (eds.) *AIST 2016. CCIS*, vol. 661, pp. 3–16. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-52920-2\\_1](https://doi.org/10.1007/978-3-319-52920-2_1)
25. Nikiforov, D.A., Korj, D.V., Sivakov, R.L.: A survey of tools for XML validation based on the object constraint language (OCL). *Inf. Technol.* **23**(5), 342–351 (2017)
26. OMG: Object Constraint Language (OCL), version 2.4. Specification, Object Management Group (2014)
27. OMG: UML Profile for National Information Exchange Model (NIEM), version 3.0. Specification, Object Management Group (2015)
28. OMG: Unified Modeling Language (UML), version 2.5. Specification, Object Management Group (2015)
29. OMG: XML Metadata Interchange (XMI), version 2.5.1. Specification, Object Management Group (2015)
30. OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation, version 1.3. Specification, Object Management Group (2016)
31. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: *EMF: Eclipse Modeling Framework 2.0*, 2nd edn. Addison-Wesley Professional, Reading (2009)

# Compositional Relational Programming with Name Projection and Compositional Synthesis

Görkem Paçacı<sup>(✉)</sup>, Steve McKeever, and Andreas Hamfelt

Department of Informatics and Media,  
Uppsala University, Uppsala, Sweden  
`gorkem.pacaci@im.uu.se`

**Abstract.** CombInduce is a methodology for inductive synthesis of logic programs, which employs a reversible meta-interpreter for synthesis, and uses a compositional relational target language for efficient synthesis of recursive predicates. The target language, Combilog, has reduced usability due to the lack of variables, a feature enforced by the principle of compositionality, which is at the core of the synthesis process. We present a revision of Combilog, namely, Combilog with Name Projection (CNP), which brings improved usability by using argument names, whilst still staying devoid of variables, preserving the compositionality.

## 1 Introduction

Automated program synthesis is the task of generating programs that follow given specifications. There are various forms of synthesis, such as *deductive synthesis*, where the specifications are expressed in a formalized language, and *inductive synthesis*, where the specifications take the form of program input/output examples [1]. These two distinct approaches have competing qualities. For example, deductive synthesis guarantees the generated program will follow the specification, which means as long as the specification is correct, the generated program will also be correct. As a result, the usability of the specification language is of crucial value. It is also an issue that the specification language has to be as expressive as the target language, and often the specification itself is as long as the program to be generated [12]. In inductive synthesis, the input/output data examples are usually provided in a very simple form, so they require almost no formality. This makes inductive synthesis more versatile in this respect, but also the correctness of the generated program will rely on the completeness of the examples given. At the end it is up to the user (or programmer) to review, and confirm/deny the generated program. This brings the usability of the target language to focus, as it becomes a determining factor of the methods success.

Here we focus on the linguistic usability of CombInduce, a method for inductive synthesis of logic programs [10, 11]. It employs a reversible meta-interpreter, which is a technique developed through the 1990s [9, 14], and recently revisited by multiple works [4, 13]. The CombInduce approach is distinguished by its

capability to synthesize two nested recursions at once, classified as a fold-2 program. Fold-2 programs include a fold operator as a recursive case of another fold operator, such as naive reversal of a list, or multiplication in Peano arithmetic. CombInduce can synthesise these using only the standard elementary predicates providing the identity and list construction, and the general list recursion operators fold-left and fold-right. This is in contrast to most recent publications on inductive synthesis, which seem to focus on only one level of recursion [4, 7, 13]. Moreover, CombInduce generates mentioned fold-2 programs in under a second with mostly less than a handful examples, compared to some other methods that require tens of examples and are distinctly slower. On the other hand, some capabilities of CombInduce are still to be investigated, such as its strength in synthesizing complex non-recursive programs, and programs with negation.

The distinguishing recursive synthesis capability of CombInduce comes at a cost. The target language, namely Combilog, is required to be compositional<sup>1</sup>, as well as being equivalent to definite clause programs in expressiveness. This means that the meaning of any operator of the language should be defined only in terms of meanings of its operands, isolating the meaning of an expression from the context it appears in. This leaves the language devoid of variables, decreasing its overall usability as a human-facing aspect of the synthesis. There are only a few notations that follow this principle. One of these is Quine’s *Predicate-Function Logic*[18], another is *Combinatory Logic* [5, 6, 19], but neither of these are intended to provide a reasonable level of usability or specialized recursion operators. As noted earlier, readability of the target language itself is crucial for the methods success, as the programmer needs to be able to comprehend and confirm the generated code as correct. If the notation is easy to modify, then when the synthesizer creates a close but incorrect candidate, the user can perform manual alterations.

In our earlier study, we identified the usability issues with Combilog to be related to the lack of variables as discussed above [16, 17]. We devised a visual language, *Visual Combilog*, which can mirror the textual Combilog code, and developed an editor that can view and edit Combilog code and Visual Combilog code side-by-side in real time, transforming back and forth as necessary. As a result of a user study involving 20 participants, we measured Visual Combilog to be significantly more usable compared to Combilog. We measured a 46% increase in speed and a 69% decrease in errors when the users were dealing with problems specifically devised to focus on argument binding [16].

As a continuation of our work on improving the linguistic usability of CombInduce, we present here a textual iteration of the Combilog language, namely, *Combilog with Name Projection* (CNP). In Sect. 2, we will reveal the specific problem that requires Combilog to be revised, and discuss why CNP is an improvement. We will continue by the formal semantics of CNP in Sect. 3. Section 4 will support our claims of usability with the results of a usability study, and Sect. 5 demonstrate how CNP is used for synthesis in place of Combilog.

---

<sup>1</sup> The concept of compositionality here refers to the *principle of compositionality* [21], where the meaning of an expression is defined as a function of meanings of its components only, and not to the concept of or-compositionality [3].

## 2 Compositional Relational Argument Binding Problem

In order to provide the mechanics for binding arguments of component predicates, Combilog devises the *make* operator which takes a source predicate and produces a new predicate where the arguments are bound to those in the source according to a given list of indices. To demonstrate, let us look at three uses of the *make* operator, in relation to how it's ordinarily achieved using variables. The first example reflects *cropping*, which eliminates arguments from a predicate:

using variables:  $p(X, Y) \leftarrow r(X, Y, \_)$       using *make*:  $p \leftarrow \text{make}([1, 2], r)$

Because the index list of the *make* operator refers only to the arguments 1 and 2, the predicate  $p$  has only two arguments bound to the respective arguments from  $r$ . The second example displays the case where the arguments in the new predicate do not appear in the same order as the source predicate. This use case of the *make* operator is referred to as *permutation* of arguments:

using variables:  $p(Y, X) \leftarrow r(X, Y, \_)$       using *make*:  $p \leftarrow \text{make}([2, 1], r)$

This is almost identical to the one in the first example, except the arguments appear in the switched order. The third argument of  $r$  is still cropped. The third and final example reflects the *expansion* use of the *make* operator. This is used to introduce new arguments that are unbound to the source predicate.

using variables:  $p(Y, X, \_) \leftarrow r(X, Y, \_)$       using *make*:  $p \leftarrow \text{make}([2, 1, 4], r)$

The only difference in this example is the introduction of a third index 4, which yields a third argument in  $p$  which is not bound to an argument of  $r$ , since  $r$  does not have a 4th argument. For introducing new unbound arguments further higher indices can be used. A complete predicate definition shows the difficulty of comprehension and modification resulting from the use of *make* more clearly. Consider this implementation of the *append* predicate in Prolog:

```
append([], Ys, Ys).
append([X|Xsrest], Ys, [X|MidList]) ← append(Xsrest, Ys, MidList).
```

In order to be able to compare the code above to its Combilog equivalent, it is useful to first observe a version of it where the syntactic sugar specific to Prolog is removed. This version of the code is given below, where the list operator  $[X|T]$  and the empty list constant  $[]$  are replaced by auxiliary predicates  $\text{cons}(H, T, [H|T])$ , and  $\text{const}[]$ :

```
append(Xs, Ys, Zs) ← const[](Xs), id(Ys, Zs).
append(Xs, Ys, Zs) ← cons(X, Xsrest, Xs) ∧
                        append(Xsrest, Ys, MidList) ∧
                        cons(X, MidList, Zs).
```

in Combilog the same predicate is written as:

$$\begin{aligned} \text{append} \leftarrow & \text{or}(\text{and}(\text{make}([1, 2, 3], \text{const}_{\square}), \text{make}([3, 1, 2], \text{id})), \\ & \text{make}([1, 2, 3], \text{and}(\text{make}([3, 4, 5, 1, 2, 6], \text{cons}), \\ & \text{make}([4, 2, 5, 6, 1, 3], \text{append}), \\ & \text{make}([4, 5, 3, 1, 6, 2], \text{cons}))). \end{aligned}$$

The Combilog definition is significantly more difficult to comprehend and modify than the Prolog equivalent. This is the main issue addressed in this paper. The intention is to introduce a new syntax for Combilog without reducing expressiveness or breaking the compositionality principle. The principle can be expressed as follows. The meaning of every valid expression in a language: (1) Should be defined as a function of meanings of its components. (2) Should not depend on the context it appears. (3) Should not depend on what comes before or after it sequentially (excluding the name-called components of the expression that may happen to come before or after it). The principle is formally stated as:

$$\llbracket \text{operator}(\varphi_1, \dots, \varphi_m) \rrbracket = F_{\text{operator}}(\llbracket \varphi_1 \rrbracket, \dots, \llbracket \varphi_m \rrbracket)$$

Free variables may incorporate the context into the meaning of an expression, therefore the compositionality principle prohibits a general use of variables. Combilog programs contain no variables, and the constructs of the language are devised in this manner. The elementary components of Combilog programs are a limited number of predicates:

- *true* for logical truth
- *const<sub>C</sub>* for introducing a constant *C*, defined as *const<sub>C</sub>(C)*.
- *id* for identity, defined as *id(X, X)*
- *cons* for working with lists, defined as *cons(H, T, [H|T])*.

Combilog programs are composed using the following operators.

- Logic operators *and* and *or*, which correspond to set intersection and union, requiring their components to have equal number of arguments:

$$\begin{aligned} \text{and}(P, Q)(X_1, \dots, X_n) & \leftarrow P(X_1, \dots, X_n) \wedge Q(X_1, \dots, X_n) \\ \text{or}(P, Q)(X_1, \dots, X_n) & \leftarrow P(X_1, \dots, X_n) \vee Q(X_1, \dots, X_n) \end{aligned}$$

- The generalized projection operator, *make*:

$$\text{make}([\mu_1, \dots, \mu_m], P)(X_{\mu_1}, \dots, X_{\mu_m}) \leftarrow P(X_1, \dots, X_n)$$

- The list recursion operators *foldr* and *foldl*:

$$\begin{aligned} \text{foldr}(P, Q)(Y, [], Z) & \leftarrow Q(Y, Z) \\ \text{foldr}(P, Q)(Y, [X|T], W) & \leftarrow \text{foldr}(P, Q)(Y, T, Z) \wedge P(X, Z, W) \\ \text{foldl}(P, Q)(Y, [], Z) & \leftarrow Q(Y, Z) \\ \text{foldl}(P, Q)(Y, [X|T], W) & \leftarrow P(X, Y, Z) \wedge \text{foldl}(P, Q)(Z, T, W) \end{aligned}$$



Let observe how these operators behave through some examples:

$$isFather \leftarrow and(isMale, hasChildren)$$

In the Combilog code above, the predicate *isFather* is defined as the conjunction of *isMale* and *hasChildren* predicates, all three predicates being unary. In contrast, in a language such as Prolog, using variables, the same could have been written as follows.

$$isFather(X) \leftarrow isMale(X) \wedge hasChildren(X)$$

When arguments are bound in a non-trivial scheme, the *make* operator is required. Let us define the *daughterOf* predicate, which succeeds if *A is the daughter of B*, A being the first argument and B the second:

$$daughterOf \leftarrow and(make([2, 1], parentOf), make([1, 2], isFemale))$$

In this example, assuming the *isFemale* is unary, but *parentOf* is binary, it is necessary to expand the *isFemale* predicate to binary by adding a second, unbound argument. The index 2 in the *make* operation associated with *isFemale* does exactly this. Since there is no second argument in the original *isFemale* predicate, *make* defines a second argument but binds it to no argument of *isFemale*, introducing an unbound argument. The same expression could have been written using variables as:

$$daughterOf(X, Y) \leftarrow parentOf(Y, X) \wedge isFemale(X)$$

As a final example of a Combilog program, let us demonstrate the recursion. The *append* example from earlier can be written using the *foldr* operator as:

$$append \leftarrow make([2, 1, 3], foldr(cons, id)).$$

Even though the definition of *append* with the *foldr* operator is quite simpler than without, the code involving a *make* operator renders any Combilog code relatively difficult to read and modify. In the next section, we present CNP which introduces argument names to overcome this usability problem.

### 3 Combilog with Name Projection (CNP)

CNP introduces the following changes to Combilog's syntax, which foremost includes the introduction of names for arguments. This allows a direct benefit since names also stand as a form of documentation as opposed to being only an identifier. There are also indirect benefits of using argument names, such as the possibility of defining operators that use argument names as a hint for schema matching between their operands' arguments. Here we summarize the overall changes CNP involves:

1. Numeric sequential argument positions (1st, 2nd, etc.) are replaced with nominal argument positions (*head*, *c*, etc.).
2. Elementary predicates are modified to include argument names.
3. The logic operators *and* and *or* are replaced with their name-aware variants which exhibit an auto-expanding behaviour, using arguments names of the components as clue.
4. The expansion use of the *make* operator is taken over by new auto-expanding logic operators (*ande* and *ore*) which accept components with any arity. This is made possible due to nominal argument positions, also.
5. A new *proj* operator is introduced, replacing the *make* operator. This new operator takes over the cropping, as well as introducing a *renaming* use.
6. The list recursion operators *foldl* and *foldr* are replaced with name-aware versions which introduce restrictions on argument names of the operands as well as fixed argument names for the resulting predicates.

These improvements are guided by a heuristic usability analysis using Green's *Cognitive Dimensions* [8], and a thorough discussion can be found in the relevant work of Paçacı [15]. They improve the usability significantly while preserving expressiveness and the compositionality principle. The resulting approach significantly resembles Codd's *Relational Algebra* [2], especially the *unordered relational domains* with the non-sequential nominal argument positions and *natural join* with the auto-expanding logic operators. The intention here is to perform logic programming rather than modelling and querying relational data. Let us observe the following examples of CNP syntax. In the examples, the argument names of a component predicate are given in a signature form  $predName : \{name1, name2, \dots\}$ . Let us start with the *daughterOf* predicate:

$$\begin{aligned}
 &parentOf : \{parent, child\} \\
 &isFemale : \{name\} \\
 &daughterOf \leftarrow ande(proj(parentOf, \{parent \mapsto parent, child \mapsto daughter\}), \\
 &\quad proj(isFemale, \{name \mapsto daughter\}))
 \end{aligned}$$

The two *proj* operators project the *parentOf* and *isFemale* predicates to produce anonymous predicates with argument names  $\{parent, daughter\}$  and  $\{daughter\}$ , respectively. Then, the *ande* operator takes their conjunction, mapping identically named arguments, producing another anonymous predicate with arguments  $\{parent, daughter\}$ , which is finally assigned to the predicate name *daughterOf*. The second example is the recursive definition of the *append* predicate:

$$append \leftarrow foldr(cons, id)$$

This example is identical to that in Combilog, since it does deal with arguments. As it will be discussed later, the *foldr* operator in CNP has fixed argument names, therefore the anonymous predicate above has the argument names inherited from this operator:  $\{as, a0, b\}$ . In order to change these to a more conventional argument names for *append*, we can project it, and assign it to a predicate name:

$$append \leftarrow proj(foldr(cons, id), \{as \rightarrow xs, a0 \rightarrow ys, b \rightarrow zs\})$$

which has the argument names  $\{xs, ys, zs\}$ .

Analogous to Combilog programs, CNP programs consist of a set of predicate definitions in the form of  $p \leftarrow \varphi$ , where  $p$  is a predicate symbol and  $\varphi$  is a body. The body is a CNP expression, constructed from elementary predicates and composition operators (*proj*, *ande*, *ore*, *foldr*, *foldl*). In the following sections, we will we will present the formal semantics of these CNP program constructs.

### 3.1 Name-Aware Tuples and Extensions

Before moving on to specific operators and their denotations, let us clarify a fundamental concept. In order to support the mechanics of argument binding with names, we shall adopt a special understanding of a relational tuple, namely,  $\alpha$ -tuple, which is in line with a *record*. An ordinary tuple with  $k$  elements can be formalized as a function from a  $k$ -size subset of the natural numbers  $K$  to elements from the Herbrand Universe,  $\tau : K \rightarrow H$ . For example, for a given tuple  $\tau = \langle t_1, \dots, t_k \rangle$ , applications of  $\tau$  as a function are  $\tau(1) = t_1, \dots, \tau(k) = t_k$ . Assuming the existence of a bijective name map  $\alpha$  from a set of names  $A$  to the same subset of the natural numbers  $K$ , that is,  $\alpha : A \rightarrow K$ , then  $\alpha$  is a compatible name map to transform an ordinary tuple  $\tau$  to an  $\alpha$ -tuple  $\tau_\alpha = \tau \circ \alpha$ . As a result,  $\tau_\alpha$  is obtained as a function from a set of names  $A$  to elements from the Herbrand universe. Given a usual tuple  $\tau = \langle t_1, \dots, t_k \rangle$  and a compatible name map  $\alpha = \{a_1 \mapsto 1, \dots, a_k \mapsto k\}$ , the applications of  $\tau_\alpha$  as a function are  $\tau_\alpha(a_1) = t_1, \dots, \tau_\alpha(a_k) = t_k$ . Since the name map  $\alpha$  is bijective, it can also be used to obtain an ordinary tuple from an  $\alpha$ -tuple, establishing the isomorphism between them. Consequently, the relational extensions discussed in the following sections should be read as sets of  $\alpha$ -tuples, rather than sets of ordinary tuples, referred to as  $\alpha$ -extensions.

In the following sections, we will give denotations of elementary predicates and operators of CNP, and we will finalize the semantics with the fixpoint semantics of CNP programs.

### 3.2 Elementary Predicates

CNP includes a set of elementary predicates as counterparts to Combilog's elementary predicates, the only difference being that they denote  $\alpha$ -extensions. The denotations of these elementary predicates are given below, with their associated name maps as subscripts to the predicate symbols.

$$\begin{aligned}
 \llbracket \text{true}_\emptyset \rrbracket &= \{\{\}\} \\
 \llbracket \text{const}(N, C)_{\alpha_1} \rrbracket &= \{\{N \mapsto C\}\} \\
 \llbracket \text{id}_{\alpha_2} \rrbracket &= \{\{a \mapsto C, b \mapsto C\} \mid C \in H\} \\
 \llbracket \text{cons}_{\alpha_3} \rrbracket &= \{\{a \mapsto X, b \mapsto Xs, ab \mapsto XXs\} \mid \\
 &\quad \langle X, Xs, XXs \rangle \in H^3 \wedge X \cdot Xs = XXs\}
 \end{aligned}$$

where

$$\begin{aligned} H &= \text{Herbrand universe of the program} \\ H^3 &= \text{tertiary cartesian product of } H \\ \alpha_1 &= \{N \mapsto 1\} \\ \alpha_2 &= \{a \mapsto 1, b \mapsto 2\} \\ \alpha_3 &= \{a \mapsto 1, b \mapsto 2, ab \mapsto 3\} \end{aligned}$$

Note that the Herbrand Universe is understood as that of a corresponding definite clause program. The *const* operator is parametric, where  $N$  is the name of the single argument of the predicate, and  $C$  is the constant the predicate should succeed for. CNP also defines an explicit instance of the *const* operator, namely *isNil*, for providing easy access to the empty list constant, defined as:

$$\text{isNil} = \text{const}(\text{nil}, [])$$

### 3.3 Projection Operator

CNP replaces Combilog's *make* operator with the *proj* operator. The projection operator  $\text{proj}(S_\alpha, P)$  produces a new predicate based on a given source predicate  $S_\alpha$ , considering the projection map given as  $P$ . A valid projection map  $P$  is a map from a non-empty set of existing names  $A \subseteq \text{Dom}(\alpha)$  from the name map of  $S$  to a set of new names  $B$ , formalized as a function as  $P : A \rightarrow B$ .  $P$  is not required to cover all names appearing in the name map of  $S$ , but has to have at least one mapping, and is required to be bijective (every old name  $a \in A$  is mapped to exactly one new name, and every new name  $b \in B$  is mapped by exactly one old name). The map entries where a name is mapped to itself by  $P$  are only *projection*, while the map entries that map names to new names are called *renaming*.

$$\begin{aligned} \llbracket \text{proj}(S_\alpha, P)_{\alpha_c} \rrbracket &= \{\tau_\alpha \circ P^{-1} \mid \tau_{\alpha_1} \in \llbracket S_\alpha \rrbracket\} \\ \text{where } \alpha_c &= \alpha \circ P^{-1} \end{aligned}$$

The resulting name map associated with application of *proj* is the composition of the name map of  $S$  and  $P^{-1}$ , containing only those names that are projected by  $P$ . The *proj* operator takes over most of the functionality of *make* except introduction of argument names, which is taken over by the logic operators, discussed in the next section.

### 3.4 Logic Operators

The logic operators defined in CNP have the option to vary their behaviour by using argument names as hints to establish bindings between arguments of the component predicates. The auto-expanding logic operators *ande* and *ore* are introduced, which behave as if the component predicates are expanded with unbound arguments to cover all argument names of all the component predicates,

taking their union. This helps to reduce the boilerplate code that emerges as a result of compulsory use of expanding the *make* operator to each operand of a logic operator, as discussed earlier. This functionally takes over the expanding use of the *make* operator in Combilog. Denotations of the auto-expanding logic operators are given below, where  $Dom(\alpha)$  refers to the domain of a name map, which consists of the relevant argument names.

$$\begin{aligned} \llbracket ande(R_{\alpha_1}, S_{\alpha_2})_{\alpha_c} \rrbracket &= \{ \tau_{\alpha_c} \in H_{\alpha_c} \mid (\tau_{\alpha_c} \supseteq \tau_{\alpha_1} \wedge \tau_{\alpha_c} \supseteq \tau_{\alpha_2}) \wedge \\ &\quad \tau_{\alpha_1} \in \llbracket R_{\alpha_1} \rrbracket \wedge \tau_{\alpha_2} \in \llbracket S_{\alpha_2} \rrbracket \} \\ \llbracket ore(R_{\alpha_1}, S_{\alpha_2})_{\alpha_c} \rrbracket &= \{ \tau_{\alpha_c} \in H_{\alpha_c} \mid (\tau_{\alpha_c} \supseteq \tau_{\alpha_1} \vee \tau_{\alpha_c} \supseteq \tau_{\alpha_2}) \wedge \\ &\quad \tau_{\alpha_1} \in \llbracket R_{\alpha_1} \rrbracket \wedge \tau_{\alpha_2} \in \llbracket S_{\alpha_2} \rrbracket \} \\ \text{where } As &= Dom(\alpha_1) \cup Dom(\alpha_2) \\ H_{As} &= \{ \{A\} \times H \mid A \in As \} \\ H_{\alpha_c} &= \{ \{T_1, \dots, T_n\} \mid T_1 \in H_{A_1} \wedge \dots \wedge T_n \in H_{A_n} \wedge \\ &\quad \{H_{A_1}, \dots, H_{A_n}\} = H_{As} \} \end{aligned}$$

The set of argument names  $A$  in the expanded composition is established as the union of all names appearing in domains of component predicates' name maps. The set  $H_A$  is a set of sets, where each set contains a mapping for one of the names in  $A$  to every element of the Herbrand Universe. Each of these sets is referred as  $T_i$  in the definition of  $H_{\alpha_c}$ . Every name-value pair  $t$  has a name from  $A$  as its first element, and an element of the Herbrand Universe as its second element. In this way,  $H_{\alpha_c}$  is established as a set of  $\alpha$ -tuples compatible with  $\alpha_c$ , which is the name map associated with the composition. The resulting name map  $\alpha_c$  maps the union of argument names in  $\alpha_1$  and  $\alpha_2$  to numeric indices, where names in  $\alpha_1$  are mapped to their original indices and the unique names in  $\alpha_2$  are mapped to the following indices, preserving their original order. For example, given  $\alpha_1 = \{a \mapsto 1, b \mapsto 2\}$  and  $\alpha_2 = \{b \mapsto 1, c \mapsto 2\}$ , the resulting name map of the composition is  $\alpha_c = \{a \mapsto 1, b \mapsto 2, c \mapsto 3\}$ .

The availability of argument names enables the implementation of a wider range of logic operators that apply various argument binding schemes. Besides *ande* and *ore*, operators such as *ando* and *oro* which bind every name-matching argument but returns a predicate with only the unmatched argument names can be implemented, resembling relation composition [20] but applying to multi-ary predicates as well as binary. Another alternative is the pair *andl/orl*, which return a predicate with only the arguments from the left-hand component. Extending these binary logic operators to multiary variants is straightforward.

### 3.5 Recursion Operators

CNP defines list recursion operators *foldr* and *foldl*. These are the counterparts to their namesakes in Combilog, and operate the same way, modulo the addition of names for their arguments. In their denotations below, the aggregate definitions *foldr* and *foldl* refer to the auxiliary definitions *foldr*<sub>0</sub>/*foldr*<sub>*i*+1</sub> and *foldl*<sub>*i*</sub>/*foldl*<sub>*i*+1</sub>, respectively, where  $i \geq 0$ .

$$\begin{aligned}
 \llbracket \text{foldr}(P, Q) \rrbracket &= \bigcup_{i=0}^{\infty} \llbracket \text{foldr}_i(P, Q) \rrbracket \\
 \llbracket \text{foldr}_0(P, Q) \rrbracket &= \{ \{ a\theta \mapsto Y, as \mapsto [], b \mapsto Z \} \in H_{\alpha_f} \mid \{ a \mapsto Y, b \mapsto Z \} \in \llbracket Q \rrbracket \} \\
 \llbracket \text{foldr}_{i+1}(P, Q) \rrbracket &= \{ \{ a\theta \mapsto Y, as \mapsto (X \cdot Xs), b \mapsto W \} \in H_{\alpha_f} \mid \\
 &\quad (\exists Z \in H \text{ s.t.} \\
 &\quad \{ a\theta \mapsto Y, as \mapsto Xs, b \mapsto Z \} \in \llbracket \text{foldr}_i(P, Q) \rrbracket \wedge \\
 &\quad \{ a \mapsto X, b \mapsto Z, ab \mapsto W \} \in \llbracket P \rrbracket) \} \\
 \llbracket \text{foldl}(P, Q) \rrbracket &= \bigcup_{i=0}^{\infty} \llbracket \text{foldl}_i(P, Q) \rrbracket \\
 \llbracket \text{foldl}_0(P, Q) \rrbracket &= \{ \{ a\theta \mapsto Y, as \mapsto [], b \mapsto Z \} \in H_{\alpha_f} \mid \{ a \mapsto Y, b \mapsto Z \} \in \llbracket Q \rrbracket \} \\
 \llbracket \text{foldl}_{i+1}(P, Q) \rrbracket &= \{ \{ a\theta \mapsto Y, as \mapsto (X \cdot Xs), b \mapsto W \} \in H_{\alpha_f} \mid \\
 &\quad (\exists Z \in H \text{ s.t.} \\
 &\quad \{ a \mapsto X, b \mapsto Y, ab \mapsto Z \} \in \llbracket P \rrbracket \wedge \\
 &\quad \{ a\theta \mapsto Z, as \mapsto Xs, b \mapsto W \} \in \llbracket \text{foldl}_i(P, Q) \rrbracket) \} \\
 \text{where } H_{\alpha_f} &= \{ \{ a\theta \mapsto A_0, as \mapsto As, b \mapsto B \} \mid \langle A_0, As, B \rangle \in H^3 \}
 \end{aligned}$$

The name maps for the fold operations are fixed, as well as their operand expressions  $P$  and  $Q$ . Operand expressions must comply with these pre-determined name maps. This is due to a design compromise for avoiding introduction of another higher-order argument (on top of  $P$  and  $Q$ ) for indicating the roles of the arguments, due to the lack of argument indices. With their fixed names, argument  $a\theta$  refers to the initial value, argument  $as$  refers to the list, and  $b$  refers to the result of the folding. The fixed name maps are as follows:  $\alpha_{\text{foldr}} = \{ a\theta \mapsto 1, as \mapsto 2, b \mapsto 3 \}$ ,  $\alpha_P = \{ a \mapsto 1, b \mapsto 2, ab \mapsto 3 \}$ ,  $\alpha_Q = \{ a \mapsto 1, b \mapsto 2 \}$ . There are also two binary variants, omitting the argument  $a\theta$ , instead obtaining the initial value through the base case. These are defined in terms of the generic *fold* operators, as in the definition of *foldr2*:

$$\text{foldr2}(P, Q) = \text{proj}(\text{foldr}(P, \text{and}e(Q, \text{id})), \{ as \mapsto as, b \mapsto b \})$$

### 3.6 Fixpoint Semantics

The model-theoretical meaning  $M_{\models}(P_{cnp})$  of a CNP program  $P_{cnp}$  is expressed in a similar way to that of Combilog as the least fixed point of the power function of an immediate consequence operator  $T^{cnp}$ . Similar to Combilog, the domain of  $T^{cnp}$  is an extension map instead of the usual Herbrand interpretations. Extension maps are structure that map predicates names to their extensions. For a CNP program  $P_{cnp}$  with  $m$  predicate definitions, the extension map is formalized as:  $E^{\alpha} = \bigcup_{i=1}^m \{ p_i \mapsto e_i^{\alpha} \}$ . Similarly, for a CNP program  $P_{cnp}$ , the immediate consequence operator  $T_{P_{cnp}}^{cnp}$  is defined using extension maps is

$T_{P_{cnp}}^{cnp}(E^\alpha) = \bigcup_i^m \{p_i \mapsto \llbracket \varphi_i \rrbracket_{E^\alpha}\}$ , where  $p_i$  refers to the  $i$ th predicate symbol,  $\varphi_i$  to the  $i$ th predicate body, and  $\llbracket \varphi \rrbracket_{E^\alpha}$  denotes the  $\alpha$ -extension of the body  $\varphi$  with regard to extension map  $E^\alpha$  and the denotations of elementary predicates. The definition of a power function of the  $T_{P_{cnp}}^{cnp}$  is given as:

$$\begin{aligned} T_{P_{cnp}}^{cnp} \uparrow 0 &= \bigcup_{j=1}^m \{p_j \mapsto \emptyset\} \\ T_{P_{cnp}}^{cnp} \uparrow (i+1) &= T_{P_{cnp}}^{cnp}(T_{P_{cnp}}^{cnp} \uparrow i) \\ T_{P_{cnp}}^{cnp} \uparrow \omega &= \bigcup_{j=1}^m \{p_j \mapsto \bigcup_{i=0}^{\infty} ((T_{P_{cnp}}^{cnp} \uparrow i)(p_j))\} \end{aligned}$$

and the model-theoretical meaning of a CNP program  $P_{cnp}$  is calculated as the least fixed point of the power function:  $M_{\models}(P_{cnp}) = T_{P_{cnp}}^{cnp} \uparrow \omega$ .

Using meaning-preserving reversible transformation steps, Combilog and CNP programs can be converted to each other, and through these transformation steps it can be proven that the least fixed point of any two program transformed would be model-theoretically isomorphic, modulo introduction/removal of names. This proof is omitted here for brevity, but it can be found in authors' other work [15]. Because the meaning of Combilog programs are proven to be equivalent to a corresponding definite clause program, the meaning of a CNP program and a corresponding definite clause program would also be isomorphic by transitivity.

## 4 Usability of CNP Programs

It is self-evident from observing the examples of CNP code in the previous section that CNP code is more readable compared to the original Combilog code. The question remains that how does it compare to notations with variables most common in the Logic Programming paradigm? In order to answer this, a within-subjects usability test has been conducted. The study compared two notations, a *Notation X* which is based on Prolog-like syntax with variables but included none of the syntactic sugar, such as list construction (`[ | ]`) or pattern matching; and a *Notation Y*, which is based on CNP. Elimination of syntactic sugar was deemed necessary to focus the study on the argument binding problem. Counter-balancing was performed by ordering the notations differently for two groups consisting of 10 participants each. Participants consisted of programmers working either in the industry or at a university, equally weighted, and distributed among the two groups. Through a pre-questionnaire, participants who had experience using Prolog were disqualified in order to have balanced results. The study was conducted through a text-based on-line questionnaire. It included six questions, out of which three involved comprehensibility and three modifiability. Each question was based on between one and four short predicate definitions. The code segments were based on working code, relating to well-known textbook

examples, but they were obfuscated differently for each notation to reduce the learning effect, and included increasing levels of complexity in terms of number of operations (variable binding or logic operators). Let us observe the first modification question, which includes the following fragments of code, given here before obfuscation of predicate names. In notation X (Prolog-like):

```
flightRoute(A, B).
trainRoute(A, B).
outInRouteOpt(D, E) :- flightRoute(D, _), trainRoute(D, _),
                        flightRoute(_, E), trainRoute(_, E).
```

and in notation Y (CNP):

```
flightRoute :: {x, y}
trainRoute  :: {x, y}
outInRouteBoth :: {a, b} =
  and(and(flightRoute {x->a}, trainRoute {x->a}),
      and(flightRoute {y->b}, trainRoute {y->b}))
```

The question required the participants to identify and refactor out a reusable component from the conjunction of `flightRoute` and `trainRoute` predicates, thereby creating a new `route` predicate. In plain text the *proj* operator is implicit, where `flightRoute {x->a}` is equivalent to *proj*(*flightRoute*, { $x \mapsto a$ }).

The participants took 276s (seconds, on average) to answer the first three comprehension questions for Notation X, while they took 345s for Notation Y, which corresponds to a 25% longer task time while using Notation Y ( $P < 0.05$ ). For the following three modification questions, the results were in favour of Notation Y. While the participants took 468 s to finish modification questions in Notation X, they took 366s for Notation Y, which corresponds to 22% shorter task time while using Notation Y ( $P < 0.05$ ). The correctness of participants' answers were also measured. For comprehension questions there were no noticeable differences. For modification questions, the participants gave 42% more correct answers while they were using Notation Y ( $P < 0.01$ ). This difference is mostly due to a single refactoring question which required alpha-renaming in Notation X but was a simple replacement in Notation Y. In a post-test questionnaire, the participants were asked three questions about their preferences. In answer to the question "Which notation did you find easier to read", 12 participants replied notation X, and 8 replied notation Y. The second question was "Which notation did you find easier to modify?", to which 8 participants replied X, 11 replied Y, and 1 replied no preference. The third question was "Which notation would you choose, if you had to use one for a project?", to which 8 participants replied X, 10 replied Y, and 2 replied no preference.

## 5 Compositional Synthesis

The synthesis approach in CombInduce can be described as a top-down search procedure that attempts to place language operators and elementary predicates



in an expression tree, written as a reversed meta-interpreter in Prolog [9,11]. The synthesizer incorporates well-modedness constraints to make sure the generated programs are procedurally terminating, and operators are utilized in valid combinations [10]. The CNP synthesizer is written the same way, in Prolog, as a single predicate *synInc* as the entry point [15]. This predicate gradually increases the depth of the search, to find shorter programs first, if there are any, while also changing the default depth-first search strategy of Prolog for a breadth-first search. To demonstrate the technique through examples, let us synthesize the naive reverse operation using the CNP synthesizer. This operation requires two levels of recursion, and an elementary predicate to construct a unit list, a list consisting of one element. Let us start by synthesizing this predicate first.

```
?- synInc(P, [a:in, aList:out], [[a:1, aList:[1]]]).
```

The call above asks for a predicate  $P$  with two arguments  $\{a, aList\}$ , and specifies a mode  $\{a : in, aList : out\}$  for the program that the synthesizer should guarantee it will procedurally terminate when executed. The call also includes a single input/output example, specifying when the argument  $a$  is 1, the argument  $aList$  should be a unit list [1]. The first predicate suggested is the correct one:

```
P = proj(ande(cons, proj(isNil, [nil->b])), [a->a, ab->aList]).
```

The suggestion constructs a conjunction of *cons* and *isNil*, binding the tail of a list to the empty list, its head to an argument  $a$ , and the whole list to the argument  $aList$ . After the user inspects and confirms the code above, and assigns a predicate name *asList*, it is available as background knowledge to the synthesizer. The next step is to synthesize the recursion stage:

```
?- synInc(P, [as:in, bs:out], [[as:[1,2,3], bs:[3,2,1]]]).
```

The call requests a predicate  $P$  with two arguments  $\{as, bs\}$ , with a single input/output example. The first suggested implementation is the correct one:

```
P = proj(foldr2(
  proj(foldr(cons, proj(asList, [a->a, aList->b])),
    [a0->a, as->b, b->ab]),
  proj(isNil, [nil->b])), [as->as, b->bs]).
```

The suggested predicate uses two nested *fold* variants, traversing a given list  $as$  (the *foldr2* operation) where each element is appended to a running list (the *foldr* operation). The *foldr* operation is a variant of the ordinary *append* predicate visited earlier in Sect. 2 with the second argument being a single element instead of a list. After the user confirms the synthesized predicate as the correct implementation, and manually assigns it a predicate name (**reverse**), it can be tested through the meta-interpreter predicate named *cnp*:

```
?- cnp(reverse, [as:[a,b,c,d], bs:Bs]).
```

succeeds with the answer binding the `Bs` variable:

```
Bs = [d,c,b,a].
```

Even though the program was asked to terminate in one direction, it can be used to un-reverse a list:

```
?- cnp(reverse, [as:As, bs:[d,c,b,a]]).
```

which succeeds with the following answer, binding the `As` variable instead:

```
As = [a,b,c,d].
```

## 6 Conclusion

In the introduction we drew attention to the usability of the target language, as it is necessary that the user inspects, comprehends and confirms the synthesized programs. If necessary, the user may choose to add more example cases that are not covered to achieve more specific programs. We have demonstrated this through a synthesis application in Sect. 5 through the synthesis of a naive reverse predicate, where the user had to intervene to initiate and confirm two separate predicates. Besides being readable, if the notation is modifiable, the user may manually alter the program as well. Among the results of the user study presented in Sect. 4, it was shown that the argument binding using nominal projection compares well to use of variables when measured in isolation. It shall be made clear that we do not claim to have devised a language which is equally user-friendly as Prolog. The user study measured only specific aspects of the languages, narrowly focusing on argument binding comprehensibility and modifiability. Programming for general problem solving involves various other skills. It is important to consider the context of our work, CNP and Combilog are fundamentally different to languages such as Prolog or Mercury, due to compositionality. A fair comparison can only be made with the likes of Predicate-Function Logic or Combinatory Logic, which are also variable-free, compositional systems of logic. For future work, we intend to improve the synthesis for even deeper levels of recursion, such as fold-3 programs, which would be able to synthesise programs such as *exponent* in Peano arithmetic.

## References

1. Basin, D., Deville, Y., Flener, P., Hamfelt, A., Fischer Nilsson, J.: Synthesis of programs in computational logic. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 30–65. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-25951-0\\_2](https://doi.org/10.1007/978-3-540-25951-0_2)
2. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)

3. Comini, M., Levi, G., Meo, M.C.: A theory of observables for logic programs. *Inf. Comput.* **169**(1), 23–80 (2001)
4. Cropper, A., Tamaddoni-Nezhad, A., Muggleton, S.H.: Meta-interpretive learning of data transformation programs. In: Inoue, K., Ohwada, H., Yamamoto, A. (eds.) *ILP 2015. LNCS (LNAI)*, vol. 9575, pp. 46–59. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-40566-7\\_4](https://doi.org/10.1007/978-3-319-40566-7_4)
5. Curry, H.B.: *Grundlagen der kombinatorischen logik*. *Am. J. Math.* **52**(3), 509–536 (1930)
6. Curry, H.B., Feys, R.: *Combinatory Logic. Studies in logic and the foundations of mathematics*, vol. 1. North-Holland Publishing Company, Amsterdam (1958)
7. Feser, J.K., Chaudhuri, S., Dillig, I.: Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.* **50**(6), 229–239 (2015)
8. Green, T.R.G.: Cognitive dimensions of notations. In: *People and Computers V*, pp. 443–460 (1989)
9. Hamfelt, A., Nilsson, J.F.: Inductive metalogic programming. In: *Proceedings Fourth International Workshop on Inductive Logic Programming*, pp. 85–96. Bad Honnef/Bonn GMD-Studien Nr. 237 (1994)
10. Hamfelt, A., Nilsson, J.F.: Inductive logic programming with well-modedness constraints. In: Rached, E. (ed.) *Proceedings of the 8th International Workshop on Functional and Logic Programming*, pp. 220–231. Centre National de la Recherche Scientifique, Institut National Polytechnique de Grenoble, Universit Joseph Fourier, Laboratoire Leibniz, Institut IMAG, 1999. UMR no 5522 (1999)
11. Hamfelt, A., Nilsson, J.F.: Inductive synthesis of logic programs by composition of combinatory program schemes. In: Flener, P. (ed.) *LOPSTR 1998. LNCS*, vol. 1559, pp. 143–158. Springer, Heidelberg (1999). [https://doi.org/10.1007/3-540-48958-4\\_8](https://doi.org/10.1007/3-540-48958-4_8)
12. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*, New York, pp. 407–426. ACM (2013)
13. Muggleton, S.H., Lin, D., Tamaddoni-Nezhad, A.: Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. *Mach. Learn.* **100**(1), 49–73 (2015)
14. Numao, M., Shimura, M.: Combinatory logic programming. In: Bruynooghe, M. (ed.) *Proceedings of the 2nd Workshop on Meta-programming in Logic*, pp. 123–136. K.U. Leuven, Belgium (1990)
15. Paçacı, G.: Representation of compositional relational programs. Ph.D. thesis, Uppsala University, Information Systems (2017)
16. Paçacı, G., Hamfelt, A.: Colour beads visual representation of compositional relational programs. In: *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, San Jose, CA, USA, pp. 131–134 (2013)
17. Paçacı, G., Hamfelt, A.: A visual system for compositional relational programming. In: *Proceedings of the The 23rd European Japanese Conference On Information Modelling And Knowledge Bases (EJC)*, Nara, Japan, 2013, pp. 235–243. IOS Press (2014)
18. Quine, W.V.: Predicate-functor logic. In: Fenstad, E. (ed.) *Proceedings of Second Scandinavian Logic Symposium*, pp. 309–315. North-Holland (1971)

19. Schönfinkel, M.: über die bausteine der mathematischen logik. *Math. Ann.* **92**(3–4), 305–316 (1924)
20. Tarski, A.: On the calculus of relations. *J. Symb. Log.* **6**(03), 73–89 (1941)
21. van Benthem, J., Ter Meulen, A.: *Handbook of Logic and Language*. Elsevier, Amsterdam (1996)

# WhaleProver: First-Order Intuitionistic Theorem Prover Based on the Inverse Method

Vladimir Pavlov<sup>(✉)</sup> and Vadim Pak

Peter the Great St. Petersburg Polytechnic University,  
29 Politechnicheskaya St., St. Petersburg 195251, Russia  
vlapav239@gmail.com, vadimpak917@gmail.com

**Abstract.** The first-order intuitionistic logic is a formal theory from the family of constructive theories. In intuitionistic logic, it is possible to extract a particular example  $x = a$  and a proof of a formula  $P(a)$  from a proof of a formula  $\exists x P(x)$ . Thanks to this feature, intuitionistic logic has many applications in mathematics and computer science. Several modern proof assistants include automated tactics for the first-order intuitionistic logic which could simplify the task of solving challenging problems, e.g. formal verification of software, hardware, and protocols.

In this article, we present a new theorem prover (named WhaleProver) for full first-order intuitionistic logic. Testing on the ILTP benchmarking library has shown that WhaleProver performance is comparable with state-of-the-art intuitionistic provers. Our prover has solved more than 800 problems from the ILTP version 1.1.2. Some of them are intractable for other provers.

WhaleProver is based on the inverse method proposed by S. Yu. Maslov. We introduce an intuitionistic inverse method calculus which is in turn a special kind of sequent calculus. Then, we describe how to adopt for this calculus several existing proof search strategies which were proposed for different logical calculi by S. Yu. Maslov, V.P. Orevkov, A.A. Voronkov, and other authors. In addition, we suggest new proof search strategy that allows avoiding redundant inferences. This article includes results of experiments with WhaleProver on the ILTP library. We believe that WhaleProver can be used further as a test bench for different inference procedures and strategies, as well as for educational purposes.

**Keywords:** Automated reasoning · Proof assistants  
Theorem proving · Inverse method · Intuitionistic logic · ILTP

## 1 Introduction

Automated theorem proving (ATP) systems are used for automated proof search in different formal theories. Nowadays, ATP systems are applied successfully to solve the variety of problems in science and industry. Those problems are traditionally considered as intellectual: mathematical theorem proving, software, hardware, and network protocols verification, software synthesis, and others.

Current interactive ATP systems (“proof assistants”) such as Coq, Agda, and Nuprl provide strong instruments for solving such problems in semi-automated mode. All the systems mentioned above are based on constructive higher-order theories (namely, on type theories). Due to the theories constructiveness, these ATP systems are able to generate programs from proofs they have built [1]. Moreover, each of them includes the programming language which provides the possibility for correct-by-construction programs synthesis.

So, constructive theories are important in ATP because every proof in these theories provides construction. The first-order intuitionistic logic is one of these theories. Standard propositional connectives and quantifiers are interpreted there in another way than in the classical logic. In particular, the classical laws of tertium non datur  $A \vee \neg A$  and double negation elimination  $(\neg\neg A) \supset A$  are unacceptable in the intuitionistic logic.

Interactive systems Coq and Nuprl can solve complex problems thanks to tactics mechanisms which include automated tactics for the first-order intuitionistic logic. For example, the automated ATP system JProver [21] is embedded in Coq, Nuprl, and MetaPRL. But, we cannot say that existing intuitionistic tactics are very effective. As a proof, we can refer to the results of JProver on the ILTP library [24].

In this paper, we present the results of the work which aim is the ATP system development. The system should exceed JProver in effectiveness. Also, it should solve new problems compared with all the current ATP systems for the first-order intuitionistic logic. We use the ILTP library [20,24] for testing and estimating performance. That library includes a wide problems collection (more than 2500) for testing and comparing ATP systems for intuitionistic logic.

The previous results of this work are presented in [18] and (in Russian) [19].

## 2 The Inverse Method

To solve the task, we choose Maslov’s inverse method [8], see also [2,7,15]. The main particularities of this method are the following: first, the proof search in the top-down direction, i.e. from axioms to the goal, and second, the subformula property of logical calculi usage.

It is hardly possible to implement an efficient theorem prover without using proof search strategies which allow reducing proof search space. Different strategies for the inverse method are analyzed in [2,9,11,13,14,23,25].

We can emphasize the two programs from existing inverse method implementations for the first-order intuitionistic logic: Gandalf [23] and Imogen [13]. The former can produce incorrect results according to the ILTP site [24]. The latter is rather efficient, mainly because of the strategies used (namely, focusing and polarization). However, that prover does not include a variety of strategies exposed in [2,11,14,23,25] which can be not less efficient.

In this work, we attempted to diminish the gap between theory and practice described above. In the paper, the inverse method calculi in the form of special sequent calculi the same as in [2] are considered. Strategies proposed by different

authors for classical and non-classical inverse method calculi have been adapted for those calculi. Also, new strategies have been developed.

### 3 Main Definitions and Notational Convention

In the paper, we use the standard first-order logic language with logical operations symbols  $\neg, \vee, \wedge, \supset$ , quantifiers  $\exists, \forall$ , predicate symbols  $P, Q, R$ , variables  $x, y, z$ , and so on, certain terms symbols  $r, s, t$ , certain formulas symbols  $A, B, C$ , and so on. Mainly, we will use terminology from [2].

**Multiset** is the set generalization which permits several occurrences of the same element. Arbitrary multisets (maybe empty) are denoted as  $\Gamma, \Delta$ . Subscripts or dashes are permitted. The note  $\Gamma, \Delta$  (or  $\Gamma, A$ ) means that it is a multiset which is obtained as a sum of  $\Gamma$  and  $\Delta$  ( $\Gamma$  and element  $A$  accordingly).

**Expression** is a certain term, formula, substitution, formulas multiset, sequent, and so on. The  $var(E)$  note means the set of all the variables of the expression  $E$ ;  $free(E)$  is the set of all free variables of  $E$ .

**Domain** of the substitution  $\theta = \{x_1/t_1, \dots, x_n/t_n\}$  marked as  $dom(\theta)$  is the set of variables  $\{x_1, \dots, x_n\}$ . The  $ran(\theta)$  note means the terms set  $\{t_1, \dots, t_n\}$ ;  $vran(\theta)$  is the set of all the variables in the terms from  $ran(\theta)$ . The  $\varepsilon$  symbol means the empty substitution.

The **substitution**  $\theta = \{x_1/t_1, \dots, x_n/t_n\}$  **narrowing** on the variables set  $\Omega$  is the substitution denoted as  $\theta|_{\Omega}$  and consisting of those and only those pairs  $x_i/t_i$  from  $\theta$  for which  $x_i \in \Omega$ . The  $\theta_{-x}$  means the  $\theta|_{dom(\theta) \setminus \{x\}}$  substitution.

For the expression  $E$  and the substitution  $\theta$  admissible for  $E$ ,  $E \cdot \theta$  is an acronym for  $E(\theta|_{free(E)})$ . If the variable  $x$  is written decisive in the expression  $E(x)$ , then  $E(t)$  means  $E\{x/t\}$  under the admissibility of  $\{x/t\}$  for  $E$ .

**Renaming** is a substitution as one-to-one mapping from its domain to itself.

The **most general unifier for substitutions**  $\sigma_1$  and  $\sigma_2$  is the most general unifier for ordered collections  $(x_1\sigma_1, \dots, x_n\sigma_1)$  and  $(x_1\sigma_2, \dots, x_n\sigma_2)$  where  $\{x_1, \dots, x_n\} = dom(\sigma_1) \cup dom(\sigma_2)$ .

The **subformula occurrence** in the formula and the **subformula occurrence polarity** (positive or negative) are defined in the standard way (see e.g. [2] or [8]). E.g. the formula  $P(x) \supset (P(y) \supset P(x))$  contains one (negative) occurrence of subformula  $P(y)$  and two occurrences of subformula  $P(x)$ , the first of which is negative, and the second is positive.

Let the  $\odot$  symbol mean any of the following symbols:  $\neg, \wedge, \vee, \supset, \forall$ , and  $\exists$ . The  $\odot$  symbol occurrence in the formula  $F$  is called the **occurrence of the kind**  $\odot^+$  ( $\odot^-$ ) if it is a positive (accordingly, negative) occurrence in  $F$ .

**Rectified formula** is the formula  $F$  where all the quantifiers bind different variables, and no bound variable occurs free in  $F$ .

**Agreement 1.** Further, the  $\xi$  symbol marks closed rectified first-order logic formula only consisting of the  $\neg, \wedge, \vee$ , and  $\supset$  logical operations symbols, and the  $\exists, \forall$  quantifiers.

The notions defined further depend on the particular formula  $\xi$ , that is why we write decisive the  $\xi$  symbol in the terms.

So,  $\xi$ -**atom** is the atomic subformula occurrence in  $\xi$ ;  $\xi$ -**sequent** is a sequent of the specific kind:  $A_1 \cdot \sigma_1, \dots, A_n \cdot \sigma_n \vdash B_1 \cdot \delta_1, \dots, B_m \cdot \delta_m$  where all  $A_i$ ,  $i \in \{1, \dots, n\}$  are negative subformulas occurrences in  $\xi$ ; all  $B_j$ ,  $j \in \{1, \dots, m\}$  are positive subformulas occurrences in  $\xi$ ;  $\sigma_i$ ,  $i \in \{1, \dots, n\}$ , and  $\delta_j$ ,  $j \in \{1, \dots, m\}$  are arbitrary substitutions of terms for variables. Part of the sequent  $S$  to the left of the symbol  $\vdash$  is called the **antecedent** of  $S$ , part of  $S$  to the right of the symbol  $\vdash$  is called the **succedent** of  $S$ .

The  $\xi$ -**formula** is a formula of the form  $\psi \cdot \sigma$  where  $\psi$  is a subformula occurrence in  $\xi$ , and  $\sigma$  is substitution of terms instead of variables. An antecedent and succedent of a  $\xi$ -sequent are  $\xi$ -formulas multisets. Further, we use the term “sequent” instead of “ $\xi$ -sequent” if it is clearly from the context that the latter sequent is meant.

The  $free(S)$  means the **set of free variables** in  $\xi$ -sequent  $S$ :  
 $free(A_1\sigma_1) \cup \dots \cup free(A_n\sigma_n) \cup free(B_1\delta_1) \cup \dots \cup free(B_m\delta_m)$ .

For  $\xi$ -sequent  $S = (A_1 \cdot \sigma_1, \dots, A_n \cdot \sigma_n \vdash B_1 \cdot \delta_1, \dots, B_m \cdot \delta_m)$  and substitution  $\theta$ , notation  $S\theta$  means the  $\xi$ -sequent  
 $A_1 \cdot (\sigma_1\theta), \dots, A_n \cdot (\sigma_n\theta) \vdash B_1 \cdot (\delta_1\theta), \dots, B_m \cdot (\delta_m\theta)$ .

The **main instance** of the  $\xi$ -sequent  $S$  is a sequent  
 $A_1\sigma_1, \dots, A_n\sigma_n \vdash B_1\delta_1, \dots, B_m\delta_m$ . Actually, a  $\xi$ -sequent and its main instance represent the same sequent in two different forms.

Let  $S$  and  $S'$  be  $\xi$ -sequents, and  $(\Gamma \vdash \Delta)$  and  $(\Gamma' \vdash \Delta')$  be their main instances correspondingly. We write  $S \subseteq S'$  if  $\Gamma \subseteq \Gamma'$  and  $\Delta \subseteq \Delta'$ . In this case,  $S$  is the **subsequent** of  $S'$ .

The  $norm(S)$  note means the **normal form** of the  $\xi$ -sequent  $S$ . It is obtained from  $S$  by means of domain narrowing for all substitutions in it: each  $\xi$ -formula  $\psi \cdot \sigma$  is substituted by  $\psi \cdot \sigma|_{free(\psi)}$ .

The  $\xi$ -formula  $\psi \cdot \sigma$  is called **proper** if  $dom(\sigma) = free(\psi)$  and  $vran(\sigma) \cap var(\xi) = \emptyset$ . The  $\xi$ -sequent  $S$  is called **proper** if each  $\xi$ -formula in it is proper.

We will say that a **formula  $\xi$  is derivable** in a sequent calculus  $C$  if a sequent consisting of the only formula  $\xi$  in the succedent is derivable in  $C$ .

Also we use standard notions related to sequent calculi, e.g. the notion of the **side formula**. For detailed background see for example [5].

Sequent calculus is called **multi-succedent** if sequents with more than one formula in the succedent could be derived in this calculus. In a **single-succedent** calculus such sequents are not derivable.

## 4 Multi-Succedent Inverse Method Calculus for the First-Order Intuitionistic Logic

Let us define a free-variable inverse method calculus  $\mathbf{IM}_{inv}^\xi$  for the first-order intuitionistic logic. The calculus  $\mathbf{IM}_{inv}^\xi$  is built exclusively for each goal formula  $\xi$ , and all formulas in the proof in this calculus are subformulas of  $\xi$ .

Axioms and inference rules of  $\mathbf{IM}_{inv}^\xi$  are presented in Table 1. In all the rules except  $(Rn)$  and  $(Nrm)$ , premises are proper  $\xi$ -sequents and do not contain



**Table 1.** Calculus  $\mathbf{IM}_{\text{inv}}^\xi$ 

$$\begin{array}{c}
\frac{}{P_1 \cdot \rho_1 \tau \vdash P_2 \cdot \rho_2 \tau} \quad (Ax) \\
\frac{\Gamma, A_1 \cdot \sigma \vdash \Delta}{\Gamma, A_1 \wedge A_2 \cdot \sigma \vdash \Delta} \quad (L\wedge_1) \qquad \frac{\Gamma, A_2 \cdot \sigma \vdash \Delta}{\Gamma, A_1 \wedge A_2 \cdot \sigma \vdash \Delta} \quad (L\wedge_2) \\
\frac{\Gamma_1 \vdash \Delta_1, A_1 \cdot \sigma_1 \quad \Gamma_2 \vdash \Delta_2, A_2 \cdot \sigma_2}{\Gamma_1 \theta, \Gamma_2 \theta \vdash \Delta_1 \theta, \Delta_2 \theta, A_1 \wedge A_2 \cdot \sigma_1 \theta} \quad (R\wedge) \\
\frac{\Gamma_1, A_1 \cdot \sigma_1 \vdash \Delta_1 \quad \Gamma_2, A_2 \cdot \sigma_2 \vdash \Delta_2}{\Gamma_1 \theta, \Gamma_2 \theta, A_1 \vee A_2 \cdot \sigma_1 \theta \vdash \Delta_1 \theta, \Delta_2 \theta} \quad (L\vee) \\
\frac{\Gamma \vdash \Delta, A_1 \cdot \sigma}{\Gamma \vdash \Delta, A_1 \vee A_2 \cdot \sigma} \quad (RV_1) \qquad \frac{\Gamma \vdash \Delta, A_2 \cdot \sigma}{\Gamma \vdash \Delta, A_1 \vee A_2 \cdot \sigma} \quad (RV_2) \\
\frac{\Gamma_1 \vdash \Delta_1, A_1 \cdot \sigma_1 \quad \Gamma_2, A_2 \cdot \sigma_2 \vdash \Delta_2}{\Gamma_1 \theta, \Gamma_2 \theta, A_1 \supset A_2 \cdot \sigma_1 \theta \vdash \Delta_1 \theta, \Delta_2 \theta} \quad (L\supset) \\
\frac{\Gamma, A_1 \cdot \sigma \vdash}{\Gamma \vdash A_1 \supset A_2 \cdot \sigma} \quad (R\supset_1) \qquad \frac{\Gamma \vdash \Delta, A_2 \cdot \sigma}{\Gamma \vdash \Delta, A_1 \supset A_2 \cdot \sigma} \quad (R\supset_2) \\
\frac{\Gamma, A_1 \cdot \sigma_1 \vdash A_1 \supset A_2 \cdot \sigma_2}{\Gamma \theta \vdash A_1 \supset A_2 \cdot \sigma_2 \theta} \quad (R\supset_3) \\
\frac{\Gamma \vdash \Delta, A \cdot \sigma}{\Gamma, \neg A \cdot \sigma \vdash \Delta} \quad (L\neg) \qquad \frac{\Gamma, A \cdot \sigma \vdash}{\Gamma \vdash \neg A \cdot \sigma} \quad (R\neg) \\
\frac{\Gamma, A \cdot \sigma \vdash \Delta}{\Gamma, \forall x A \cdot \sigma_{-x} \vdash \Delta} \quad (L\forall) \qquad \frac{\Gamma \vdash A \cdot \sigma}{\Gamma \vdash \forall x A \cdot \sigma_{-x}} \quad (R\forall) \\
\frac{\Gamma, A \cdot \sigma \vdash \Delta}{\Gamma, \exists x A \cdot \sigma_{-x} \vdash \Delta} \quad (L\exists) \qquad \frac{\Gamma \vdash \Delta, A \cdot \sigma}{\Gamma \vdash \Delta, \exists x A \cdot \sigma_{-x}} \quad (R\exists) \\
\frac{\Gamma, A \cdot \sigma_1, A \cdot \sigma_2 \vdash \Delta}{\Gamma \theta, A \cdot \sigma_1 \theta \vdash \Delta \theta} \quad (LC) \qquad \frac{\Gamma \vdash \Delta, A \cdot \sigma_1, A \cdot \sigma_2}{\Gamma \theta \vdash \Delta \theta, A \cdot \sigma_1 \theta} \quad (RC) \\
\frac{S}{S\rho} \quad (Rn) \qquad \frac{S}{\text{norm}(S)} \quad (Nrm)
\end{array}$$

common free variables. In the axiom scheme  $(Ax)$ ,  $P_1, P_2$  are  $\xi$ -atoms;  $\rho_1, \rho_2$  are renamings for which  $\text{free}(P_1\rho_1) \cap \text{free}(P_2\rho_2) = \emptyset$ ;  $\tau$  is the most general unifier for formulas  $P_1\rho_1$  and  $P_2\rho_2$ . The  $\theta$  substitution is the most general unifier for  $\sigma_1$  and  $\sigma_2$ . In the rule  $(Rn)$ ,  $S$  is a  $\xi$ -sequent,  $\rho$  is a renaming. In the rule  $(Nrm)$ ,  $S$  is a  $\xi$ -sequent. In the rules  $(L\exists)$  and  $(R\forall)$ , the eigenvariable condition holds for the term  $x\sigma$ :  $x\sigma$  is a *variable* not occurring *free* in the conclusion.

The structural rules  $(LC)$  and  $(RC)$  are called the **contraction** rules. The rules  $(Rn)$  and  $(Nrm)$  are called the **renaming** and **normalization** rules correspondingly. The calculus has been constructed using the standard recipe from [2, 10].

It can be shown that the calculus  $\mathbf{IM}_{\text{inv}}^\xi$  is equivalent to the GHPC calculus from the article by Dragalin [3] which is in turn equivalent to the HPC (Heyting's Predicate Calculus).

**Theorem 1 (Completeness of  $\mathbf{IM}_{\text{inv}}^\xi$ ).** *Let  $\xi$  be a closed rectified formula corresponding to Agreement 1, and  $\xi'$  is obtained from  $\xi$  by replacing all subfor-*

mula occurrences of the form  $\neg A$  by  $A \supset \perp$ . Then,  $\xi$  is derivable in  $\mathbf{IM}_{\text{inv}}^\xi$  iff  $\xi'$  is derivable in GHPC.

*Proof.* The proof is similar to the completeness proof for the calculus  $\mathbf{I}_{\text{inv}}^\xi$  which is given in [2].

Actually, in the axiom  $(Ax)$  of  $\mathbf{IM}_{\text{inv}}^\xi$ , it is enough to rename variables in the only one of two  $\xi$ -atoms. However, the axiom  $(Ax)$  from Table 1 has its technical advantages: for each  $\xi$ -atom, we could fix “unique” renaming and use it in all related instances of  $(Ax)$  instead of renaming this  $\xi$ -atom each time when the  $(Ax)$  axiom is applied.

The calculus  $\mathbf{IM}_{\text{inv}}^\xi$  is multi-succedent, in contrast to existing single-succedent inverse method calculi, e.g. calculus  $\mathbf{I}_{\text{inv}}^\xi$  from [2].

## 5 Proof Search Strategies

To make the proof search by the inverse method more efficient, we need to supplement the developed multi-succedent calculus  $\mathbf{IM}_{\text{inv}}^\xi$  with proof search strategies (or tactics in Maslov’s terminology) to get rid of redundant sequents or inferences. Simultaneously, we will consider the strategies applicability to the single-succedent calculus  $\mathbf{I}_{\text{inv}}^\xi$  from [2].

### 5.1 Subsumption Strategy

Let us introduce the subsumption strategy for the calculus  $\mathbf{IM}_{\text{inv}}^\xi$  which is not a standard subsumption like in [14] or [13], but rather an adaptation of a stronger form for the subsumption proposed by Voronkov in [25]. We will refer to the former subsumption form as the **standard subsumption**, and to the latter as the **strong subsumption**. A strong subsumption adaptation consists in formulation of subsumption relation for  $\mathbf{IM}_{\text{inv}}^\xi$  with specific restrictions which satisfies the general definition of subsumption relation for deductive systems from [25].

First, let us define a standard subsumption relation  $\prec$  for  $\mathbf{IM}_{\text{inv}}^\xi$ .

**Definition 1.** Relation  $S \prec S'$  holds for  $\xi$ -sequents  $S$  and  $S'$  iff there exists such a substitution  $\sigma$  that  $S'\sigma \subseteq S$ .

**Definition 2.** **Basic rules** of  $\mathbf{IM}_{\text{inv}}^\xi$  include all the rules except renaming and normalization. The same definition is also applicable to the calculus  $\mathbf{I}_{\text{inv}}^\xi$  from [2] (in that calculus, the normalization rule is used implicitly).

**Definition 3.** The renaming rule application (with renaming  $\rho$ ) for a  $\xi$ -sequent  $S$  is called **proper** iff  $\text{free}(S\rho) \cap \text{var}(\xi) = \emptyset$ . In this case,  $\rho$  is called **proper renaming** for  $S$ .

Let  $\Pi$  be a  $\xi$ -sequent  $S'$  derivation from a  $\xi$ -sequent  $S$  in  $\mathbf{IM}_{\text{inv}}^\xi$  ( $\mathbf{I}_{\text{inv}}^\xi$ ). This derivation is called **proper** iff every basic rule application in  $\Pi$  is followed by a proper renaming rule application which is in turn followed by a normalization rule application.

*Note 1.* Any derivation in  $\mathbf{IM}_{\text{inv}}^\xi$  ( $\mathbf{I}_{\text{inv}}^\xi$ ) can be transformed into a proper derivation.

**Definition 4.** Let  $\mathfrak{R}^\xi$  be the list which contains the following rules of the calculus  $\mathbf{IM}_{\text{inv}}^\xi$ :  $(L\wedge_1)$ ,  $(L\wedge_2)$ ,  $(R\vee_1)$ ,  $(R\vee_2)$ ,  $(L\neg)$ ,  $(R\supset_2)$ ,  $(L\forall)$ ,  $(R\exists)$ ,  $(L\exists)$ , plus  $(R\forall)$  in case when  $\xi$  does not contain negative occurrences of  $\vee$ .

Now we can define the binary relation  $\prec_{\text{IM}}$  on  $\xi$ -sequents (the strong subsumption relation).

**Definition 5.** Relation  $S \prec_{\text{IM}} S'$  holds (i.e.  $S'$  subsumes  $S$ ) iff there exists a  $\xi$ -sequent  $S^*$  such that  $S^* \prec S'$ , and  $S^*$  is properly derivable from  $S$  only using rules from  $\mathfrak{R}^\xi$ , and without applying the rules  $(R\vee_1)$ ,  $(R\vee_2)$ ,  $(L\neg)$ ,  $(R\supset_2)$ ,  $(R\exists)$ , and  $(R\forall)$  when the side formula in the premise has the form  $(A \supset B) \cdot \sigma$ .

**Definition 6.** The subsumption strategy for  $\mathbf{IM}_{\text{inv}}^\xi$  is the following: if two sequents  $S$  and  $S'$  have been derived in the proof search process, and  $S \prec_{\text{IM}} S'$ , then  $S$  can be removed immediately from the proof search space.

It can be shown that this subsumption strategy is complete not only for the calculus  $\mathbf{IM}_{\text{inv}}^\xi$ , but also for  $\mathbf{I}_{\text{inv}}^\xi$  from [2].

**Theorem 2 (The subsumption strategy completeness).** *The subsumption strategy is complete for  $\mathbf{IM}_{\text{inv}}^\xi$  and  $\mathbf{I}_{\text{inv}}^\xi$ .*

*Proof.* This theorem can be proved by showing that the relation  $\prec_{\text{IM}}$  is subsumption ordering for  $\mathbf{IM}_{\text{inv}}^\xi$  and  $\mathbf{I}_{\text{inv}}^\xi$  (see [25]). An alternative way is to show compatibility of the standard subsumption with the strategy of lifting the rules from  $\mathfrak{R}^\xi$  upwards in the proof tree. Permutability of rules from  $\mathfrak{R}^\xi$  can be shown in the same way as it done by Kleene in [5]. The only special case is lifting the rules  $(L\exists)$  and  $(R\forall)$ : they are not permutable with the rules  $(R\exists)$  and  $(L\forall)$  in the sense of S. Kleene, but it is permissible to apply them as soon as possible (after lifting the rule  $(L\exists)$ , a duplicate application of this rule might be needed).

## 5.2 Subsumption Strategy (Reformulation)

Let us revise Definition 5 from the previous subsection. It is not obvious how to use this definition in a practical algorithm for checking whether  $S \prec_{\text{IM}} S'$  holds for arbitrary  $\xi$ -sequents  $S$  and  $S'$ . We will formulate more practical definition of  $\prec_{\text{IM}}$ . For this purpose, we need to introduce additional definitions and agreements.

Let  $\psi$  and  $\psi'$  be subformula occurrences in  $\xi$ . Then,  $\psi \in \psi'$  means that  $\psi$  belongs to  $\psi'$ .

**Definition 7.** We will say that the **path from  $\psi'$  to  $\psi$**  (where  $\psi \in \psi'$ ) **contains an occurrence** of the logical symbol  $\odot$  iff this occurrence belongs to  $\psi'$  and does not belong to  $\psi$ . Similar definition can be given for occurrences of the type  $\odot^+$  (of the type  $\odot^-$ ) and for subformulas occurrences.

**Definition 8.** The **path from  $\psi'$  to  $\psi$**  ( $\psi \in \psi'$ ) **contains the sign change** iff it contains such occurrence of the subformula of the form  $\neg A$  or  $A \supset B$  that  $\psi$  belongs to the specified occurrence of  $A$ .

**Definition 9.** Let  $S, S'$  be  $\xi$ -sequents, and let  $F = (\psi \cdot \sigma)$ ,  $F' = (\psi' \cdot \sigma')$  be  $\xi$ -formulas from  $S$  and  $S'$  respectively. The  $F'$  formula **properly subsumes  $F$  with substitution  $\delta$**  iff the following conditions hold.

1.  $\psi \in \psi'$ .
2.  $\forall x \in \text{free}(\psi') \cap \text{free}(\psi) \ x\sigma'\delta = x\sigma$ .
3. Let  $x$  be an arbitrary variable from  $\text{free}(\psi') \setminus \text{free}(\psi)$ , and  $\rho$  be proper renaming for  $S$ . Then,  $x\sigma'\rho$  is a variable which only once occurs in all substitutions from  $S'\rho$ .
4. A path from  $\psi'$  to  $\psi$  only contains occurrences of type  $\wedge^-, \vee^+, \neg^-, \supset^+, \forall^-, \exists^+, \exists^-$ , plus  $\forall^+$  if  $\xi$  does not contain negative occurrences of  $\vee^-$ .
5. A path from  $\psi'$  to  $\psi$  contains at most one occurrence of type  $\supset^+$ .
6. If a path from  $\psi'$  to  $\psi$  contains a positive occurrence of subformula of the form  $A \supset B$ , then this occurrence coincides with  $\psi'$  and  $\psi \in B$ .
7. If  $\psi$  has the form  $A \supset B$ , then  $\psi'$  coincides with  $\psi$ .

The third condition from Definition 9 ensures that all free variables from  $\psi'$  which does not occur in  $\psi$  are substituted by unique variables. It helps to avoid “collision of variables” when positive quantifier rules are applicable to  $S$ , but not applicable to  $S'$ . Conditions 4–7 correspond to the restriction on the rules from Definition 5.

**Definition 10 (Reformulation of relation  $\prec_{\text{IM}}$ ).** Let  $S$  and  $S'$  be  $\xi$ -sequents, and  $F'_1, \dots, F'_n$  be all different occurrences of  $\xi$ -formulas in  $S'$ . Then,  $S \prec_{\text{IM}} S'$  iff there exist **different** occurrences  $F_1, \dots, F_n$  of  $\xi$ -formulas in  $S$ , and a substitution  $\delta$  such that  $F'_i$  properly subsumes  $F_i$  with the substitution  $\delta$  ( $i \in \{1, \dots, n\}$ ).

### 5.3 Reduction Strategy

In this subsection, we consider the reduction strategy (the strategy named following Tammet [23]).

The reduction step consists in the following. If any rule  $R$  from  $\mathfrak{R}^\xi$  is applicable to a sequent  $S$ , then  $S$  is replaced by a result of applying  $R$  together with proper renaming to  $S$ . Reduction steps can be applied to a sequent  $S$  recursively until deriving such a sequent  $S'$  (called a **reduction** of  $S$ ) that no more reduction steps are applicable to  $S'$ .

**Definition 11.** The **reduction strategy** consists in replacing any derived sequent by its reduction.

**Theorem 3 (The reduction strategy completeness).** The reduction strategy is complete for  $\text{IM}_{\text{inv}}^\xi$  and  $\text{I}_{\text{inv}}^\xi$ .

*Proof.* It follows from the subsumption strategy completeness for  $\text{IM}_{\text{inv}}^\xi$  ( $\text{I}_{\text{inv}}^\xi$ ).

Reduction strategy for other inverse method calculi can be found in articles by Tammet [23] and Voronkov [25].

### 5.4 Trivial Contraction Strategy

By analogy with the “tactics of transition to trivial specifications” proposed by S. Yu. Maslov for the classical logic [11], we suggest to use the *trivial contraction strategy* for  $\mathbf{IM}_{\text{inv}}^\xi$  and  $\mathbf{I}_{\text{inv}}^\xi$ : if derived sequent  $S$  has such contraction  $S'$  that  $S \prec S'$ , then  $S$  is immediately replaced by  $S'$ .

The trivial contraction can be seen as the reduction strategy analog for contraction rules. The trivial contraction strategy completeness follows from the subsumption strategy completeness.

### 5.5 Removing Inadmissible Sequents Strategy

Let us first consider the inadmissible sequents class which in its general form has been noticed by Maslov [8] and specified for classical logic by Orevkov [16] (with a difference that clauses are used instead of sequents in the mentioned articles). In the case of the sequent calculi  $\mathbf{IM}_{\text{inv}}^\xi$  and  $\mathbf{I}_{\text{inv}}^\xi$ , this class includes sequents with inadmissible substitutions. For example, this class includes each sequent which contains such a  $\xi$ -formula  $\psi \cdot \sigma$  and a variable  $x \in \text{free}(\psi)$  that  $x\sigma$  is not a variable and  $x$  is bound in  $\xi$  by a quantifier of type  $\forall^+$  or  $\exists^-$ .

It can be proved that the strategy of removing sequents with inadmissible substitutions is still complete for  $\mathbf{IM}_{\text{inv}}^\xi$  and  $\mathbf{I}_{\text{inv}}^\xi$ .

**Definition 12.** *The strategy of removing sequents containing inadmissible substitutions (RSIS) for  $\mathbf{IM}_{\text{inv}}^\xi$  and  $\mathbf{I}_{\text{inv}}^\xi$  consists in removing any derived sequent which contains inadmissible substitutions.*

The next, we consider a new strategy for  $\mathbf{IM}_{\text{inv}}^\xi$  and  $\mathbf{I}_{\text{inv}}^\xi$  which can be considered as a stronger variant of nesting strategy by Tammet [23]. Our strategy is called the strategy of removing sequents with inadmissible formulas. It is based on two sets of inadmissible sequents  $U\_form_1$  and  $U\_form_2$  which will be defined below.

**Definition 13.** *A subformula  $\psi$  in formula  $\xi$  occurrence is called **strictly positive** iff  $\psi$  is a positive occurrence in  $\xi$ , and  $\psi$  does not belong to any negative subformula occurrence in  $\xi$ .*

**Definition 14.** *Let  $\psi$  and  $\psi'$  be subformulas occurrences in  $\xi$ . The  $\psi$  occurrence **critically belongs** to  $\psi'$  (denoted  $\psi \bar{\in} \psi'$ ) iff  $\psi \in \psi'$ , and there exists such positive subformula  $\psi^+$  occurrence in  $\xi$  that  $\psi^+$  has the form  $A \supset B$ ,  $\neg A$ , or  $\forall xA$ ,  $\psi^+ \in \psi'$ , and  $\psi \in A$ .*

**Definition 15.** *Let  $\psi_1$  and  $\psi_2$  be subformulas occurrences in  $\xi$ . By the **common superformula** of  $\psi_1$  and  $\psi_2$  we mean such subformula  $\psi^*$  occurrence in  $\xi$  that  $\psi_1 \in \psi^*$  and  $\psi_2 \in \psi^*$ . The **minimal common superformula** of  $\psi_1$  and  $\psi_2$  (denoted  $\text{par}^\xi(\psi_1, \psi_2)$ ) is such common superformula that  $\text{par}^\xi(\psi_1, \psi_2) \in \psi^*$  for any other common superformula  $\psi^*$  of  $\psi_1$  and  $\psi_2$ .*

**Definition 16.** A set  $U\_form_1$  for  $\mathbf{IM}_{inv}^\xi$  and  $\mathbf{I}_{inv}^\xi$  is defined as follows. A sequent  $S$  belongs to the set  $U\_form_1$  iff there exists such a pair of  $\xi$ -formulas  $\psi_1 \cdot \sigma_1$ ,  $\psi_2 \cdot \sigma_2$  from  $S$  that  $\psi_2 \not\subseteq \psi_1$  (in particular,  $\psi_2$  does not coincide with  $\psi_1$ ),  $par^\xi(\psi_1, \psi_2)$  is a strictly positive subformula occurrence in  $\xi$ , and at least one of the following conditions holds.

1.  $par^\xi(\psi_1, \psi_2)$  has the form  $A \supset B$  and coincides with  $\psi_2$ , and either  $\psi_1 \bar{\in} A$ , or  $\psi_1 \bar{\in} B$ .
2.  $par^\xi(\psi_1, \psi_2)$  has the form  $A \supset B$ ,  $\psi_1 \bar{\in} A$ ,  $\psi_2 \in B$ , and  $\psi_2$  is a strictly positive subformula occurrence in  $\xi$ .
3.  $par^\xi(\psi_1, \psi_2)$  does not have the form  $A \supset B$ , and either  $\psi_1 \bar{\in} par^\xi(\psi_1, \psi_2)$ , or  $\psi_2 \bar{\in} par^\xi(\psi_1, \psi_2)$ .

**Definition 17.** Let  $\psi$  and  $\psi'$  be subformulas occurrences in  $\xi$ . The  $\psi$  occurrence **critically belongs** to  $\psi'$  **without sign change** (denoted  $\psi \hat{\in} \psi'$ ) iff  $\psi \in \psi'$ , and there exists such positive subformula  $\psi^+ = (\forall xA)$  occurrence in  $\xi$  that  $\psi^+ \in \psi'$ ,  $\psi \in A$ , and path from  $\psi^+$  to  $\psi$  does not contain sign change.

**Definition 18.** Let us define a set  $U\_form_2$  for the calculus  $\mathbf{IM}_{inv}^\xi$ . A sequent  $S$  belongs to  $U\_form_2$  iff there exists such a pair of  $\xi$ -formulas  $\psi_1 \cdot \sigma_1$ ,  $\psi_2 \cdot \sigma_2$  from the succedent of  $S$  that for each  $i \in \{1, 2\}$   $\psi_i \hat{\in} \xi$  holds, and at least for one  $i \in \{1, 2\}$   $\psi_i \hat{\in} par^\xi(\psi_1, \psi_2)$  holds.

A set  $U\_form_2$  for the calculus  $\mathbf{I}_{inv}^\xi$  is empty. It can be shown that sequents belonging to the sets  $U\_form_1$  and  $U\_form_2$  cannot occur in a derivation of  $\xi$  in  $\mathbf{IM}_{inv}^\xi$  and  $\mathbf{I}_{inv}^\xi$ .

**Definition 19.** The **strategy of removing sequents containing inadmissible formulas (RSIF)** for  $\mathbf{IM}_{inv}^\xi$  and  $\mathbf{I}_{inv}^\xi$  consists in removing any derived sequent which belongs to  $U\_form_1$  or  $U\_form_2$ .

## 5.6 Singular Sequent Strategy for the Multi-Succedent Calculus

Since any proof in the single-succedent calculus  $\mathbf{I}_{inv}^\xi$  can be easily transformed into a proof in the multi-succedent calculus  $\mathbf{IM}_{inv}^\xi$ , we can consider the following strategy for  $\mathbf{IM}_{inv}^\xi$ .

**Definition 20.** The **singular strategy** for  $\mathbf{IM}_{inv}^\xi$  consists in constructing the formula  $\xi$  proof in the calculus  $\mathbf{I}_{inv}^\xi$  using any complete strategies for  $\mathbf{I}_{inv}^\xi$ , and then, transforming this proof into a proof in  $\mathbf{IM}_{inv}^\xi$ .

## 5.7 Combining Strategies

**Theorem 4 (The strategies compatibility).** Any combination of the following strategies is complete for  $\mathbf{IM}_{inv}^\xi$  and  $\mathbf{I}_{inv}^\xi$ : the subsumption strategy, the reduction strategy, the strategies RSIS and RSIF, the trivial contraction strategy.

*Proof.* The RSIS and RSIF strategies are compatible with other strategies because they only allow removing inadmissible sequents. Other three strategies are based on the same subsumption relation, and therefore are compatible too.

**Corollary 1.** *The singular strategy together with any combination of the strategies for  $\mathbf{I}_{\text{inv}}^\xi$  mentioned in Theorem 4 is complete for  $\mathbf{IM}_{\text{inv}}^\xi$ .*

## 5.8 Example

Let us demonstrate how some strategies work on a simple example. Let the formula  $\xi = \forall x \neg \neg (P(x) \vee \neg \bar{P}(x))$ .

We will build the proof of  $\xi$  in  $\mathbf{IM}_{\text{inv}}^\xi$  using the strategies presented above. First of all,  $\xi$  contains two atomic subformula  $P(x)$  occurrences, positive and negative. Negative occurrence is marked with overline to distinguish it in the proof. So, we could derive only one axiom (up to renaming):

$$1. \bar{P}(x) \cdot \{x/v\} \vdash P(x) \cdot \{x/v\} \quad [(Ax)].$$

Variable  $v$  is new. Now we could apply the reduction strategy to the sequent 1 using the rule  $(RV_1)$ . So, we replace sequent 1 with the following one:

$$1. \bar{P}(x) \cdot \{x/v\} \vdash P(x) \vee \neg \bar{P}(x) \cdot \{x/v\} \quad [(Ax) + (RV_1)].$$

We could continue reducing the sequent 1 using the rule  $(L\bar{\neg})$ :

$$1. \bar{P}(x) \cdot \{x/v\}, \neg (P(x) \vee \neg \bar{P}(x)) \cdot \{x/v\} \vdash \quad [(Ax) + (RV_1) + (L\bar{\neg})].$$

No further reductions are possible, but now we could apply the rule  $(R\bar{\neg})$ :

$$2. \neg (P(x) \vee \neg \bar{P}(x)) \cdot \{x/v\} \vdash \neg \bar{P}(x) \cdot \{x/v\} \quad [(R\bar{\neg}) : 1].$$

Next, we apply the reduction strategy to the sequent 2 using the rules  $(RV_2)$  and  $(L\bar{\neg})$ , and then contract two equal  $\xi$ -formulas in the antecedent using the trivial contraction strategy (let us skip intermediate steps).

$$2. \neg (P(x) \vee \neg \bar{P}(x)) \cdot \{x/v\} \vdash \quad [(R\bar{\neg}) + (RV_2) + (L\bar{\neg}) + (LC) : 1].$$

Now, we could finish the proof by applying the rules  $(R\bar{\neg})$  and  $(R\forall)$ .

$$3. \vdash \neg \neg (P(x) \vee \neg \bar{P}(x)) \cdot \{x/v\} \quad [(R\bar{\neg}) : 2].$$

$$4. \vdash \forall x \neg \neg (P(x) \vee \neg \bar{P}(x)) \cdot \varepsilon \quad [(R\forall) : 3].$$

The eigenvariable condition holds for the rule  $(R\forall)$ , since variable  $v$  does not occur free in the conclusion.

## 6 WhaleProver and Experiments on the ILTP Library

WhaleProver is a software implementation of the inverse method for intuitionistic and classical logics that has been developed within the scope of current research. WhaleProver supports several inverse method calculi and their modifications including  $\mathbf{IM}_{inv}^\xi$  from Sect. 4, plus  $\mathbf{I}_{inv}^\xi$  and  $\mathbf{C}_{inv}^\xi$  from [2]. Our prover can be configured to use any combination of strategies from Sect. 5. Also, it can be extended with new sequent calculi and proof search strategies. WhaleProver is written in C++ and uses an adapted variant of given clause algorithm [12].

We carried out several experiments with WhaleProver on the ILTP library [20] version 1.1.2. We used the computer with Intel Core i5 3.40 GHz, OC Windows 7, and 16 Gb RAM (the prover was executed on a single core in 32-bit address space, therefore, available memory size was limited by 2 Gb).

The first series of experiments was connected with comparison of strategies from Sect. 5 on the representative subset of 366 problems from the ILTP. We used the following comparison criteria: average proof length, average proof space size, and summary proof time. The experiments have shown that the strong subsumption is about 20–30 % more efficient (by all three criteria) than the standard subsumption. The subsumption and reduction strategies have the most impact on the prover performance. New RSIF strategy helps to reduce proof time by 25 %. The singular strategy allows reducing proof time in average by 10 %, but its impact becomes less significant when all other strategies are turned on. All the strategies together allow reducing total proof time up to 30 times. We do not present here detailed results of the comparison due to the lack of space.

As the second experiment, we have run our prover on the same computer on all ILTP problems with 100s timeout for each problem. The WhaleProver results and statistics of other first-order intuitionistic provers are presented in Table 2.

**Table 2.** Comparison of WhaleProver with other provers on the ILTP

Param	JProver	ft-Prolog	ft-C	ileanSeP	ileanTAP	ileanCoP	Imogen	WhaleProver
Timeout used, s	600	600	600	600	600	600	600	<b>100</b>
Solved after 1 s	243	285	351	249	299	557	681	<b>660</b>
Solved after 10 s	254	294	357	282	307	603	731	<b>719</b>
Solved after 100 s	262	295	364	301	312	647	854	<b>811</b>
Total solved	268	299	364	313	315	690	856	<b>811</b>
Total proved	264	299	334	309	311	610	645	<b>620</b>
Total refuted	4	0	30	4	4	80	211	<b>191</b>



WhaleProver results were obtained in compliance with the guidelines for use of the ILTP [20]. The only changes made to the original problems were syntax transforming and equality axioms adding by using the tptp2X tool.

The results of Imogen prover were taken from [13], and also, from the regression statistics provided with the source code of the prover [4]. Results of other provers are published on the ILTP website [24]. Despite the fact that different provers were launched on computers with different configuration, comparison from Table 2 is correct according to the methodology recommended by ATP experts [22]: all compared provers have passed their PPP (Peter Principle Point).

We are aware of the recent testing results of the new nanoCop-i prover (700 problems from the ILTP proved) and the new version of ileanCoP prover – ileanCoP 1.2 (717 problems proved in “full” configuration) published by Otten [17]. However, we did not include those results to Table 2 since not all details needed to fill in this table are available. Also, it is not clear whether those provers have passed their PPP or not, and how many problems can be refuted by them.

At least, we can draw the conclusion that WhaleProver solves about three times more problems than JProver, and that WhaleProver performance is comparable with performance of the most efficient intuitionistic provers.

WhaleProver solves 70 problems with known intuitionistic status which have not been solved by any prover included into ILTP platform (it includes all provers from Table 2 except Imogen and WhaleProver). If we take into account the regression results of Imogen from [4], then WhaleProver solves 16 new problems with known status in comparison with all other provers. Also, WhaleProver solves 16 new problems with unknown intuitionistic status (i.e. “Unsolved” or “Open”). For example, the following problems related to the natural language processing were solved (with result “not a theorem”): NLP198+1 (processing the text fragment about the old dirty white Chevy) and NLP223+1 (processing the phrase “Vincent believes that every man smokes. . .”). Moreover, WhaleProver solves a number of problems several times faster than other provers can.

## 7 Conclusion

In the current article, we discussed the inverse method application for the first-order intuitionistic logic. We presented a multi-succedent inverse method calculus  $\mathbf{IM}_{\text{inv}}^{\xi}$  which differs from the existing single-succedent intuitionistic inverse method calculi, e.g. the calculus  $\mathbf{I}_{\text{inv}}^{\xi}$  from [2]. We have shown how to adopt existing proof search strategies to the calculus  $\mathbf{IM}_{\text{inv}}^{\xi}$ , and suggested new strategy RSIF which allows removing redundant sequents from the proof search space. All considered strategies are also applicable to the calculus  $\mathbf{I}_{\text{inv}}^{\xi}$ .

All the suggested strategies were implemented in the theorem prover for the first-order intuitionistic logic called WhaleProver. Experiments on problems from the ILTP library have shown that it is possible to obtain an efficient proof search procedure by combining these strategies. WhaleProver has shown promising results on the ILTP comparable with results of state-of-the-art intuitionistic provers, e.g. ileanCoP and Imogen. We are pretty sure that it is possible

to improve our prover performance further by extending it with new powerful strategies and heuristics, e.g. by adopting focusing strategy from [13]. However, to our knowledge, the problem of developing complete combination of focusing and strong subsumption strategies has not been solved yet.

In the future, we plan to use WhaleProver as a test bench for inverse method calculi and strategies. Also, we are going to integrate WhaleProver as a plug-in to existing proof assistants (Coq, Nuprl), following works [6, 21].

## References

1. Constable, R. L.: On building constructive formal theories of computation. Noting the roles of Turing, Church, and Brouwer. In: Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science (LICS 2012), pp. 2–8 (2012)
2. Degtyarev, A., Voronkov, A.: The inverse method. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, vol. 1, pp. 179–272. Elsevier, Amsterdam (2001)
3. Dragalin, A.G.: Mathematical intuitionism. Introduction to proof theory (Russian). Nauka, Moscow (1979). English transl. in Translations of Mathematical Monographs 67 (1988)
4. Imogen GitHub page. <https://github.com/seanmcl/imogen>
5. Kleene, S.C.: Permutability of inferences in Gentzen’s calculi LK and LJ. Mem. Amer. Math. Soc. **10**, 1–26 (1952). Russian transl. in “Matematicheskaya teoriya logicheskogo vyvoda”, 208–236 “Nauka”, Moscow (1967)
6. Kunze, F.: Towards the integration of an intuitionistic first-order prover into Coq. In: Proceedings of the 1st International Workshop Hammers for Type Theories (HaTT 2016) (2016). <https://arxiv.org/abs/1606.05948>
7. Lifschitz, V.: What is the inverse method? J. Autom. Reasoning **5**(1), 1–23 (1989)
8. Maslov, S.Y.: The inverse method for establishing deducibility for logical calculi (Russian). Trudy Matem. Inst. AN SSSR **98**, 26–87 (1968). English transl. in Proc. Steklov Inst. of Mathematics 98, 25–95, AMS, Providence (1971)
9. Maslov, S.Y.: Deduction-search tactics based on unification of the order of members in a favourable set (Russian). Zap. Nauchn. Sem. LOMI **16**, 126–136 (1969). English transl. in Seminars in Mathematics. Steklov Math. Inst. 16, Consultants Bureau, New York - London (1971)
10. Maslov, S.Y.: Deduction search in calculi of general type (Russian). Zap. Nauchn. Sem. LOMI **32**, 59–65 (1972). English transl. in Journal of Soviet Mathematics, vol. 6, no. 4, 395–400 (1976)
11. Maslov, S.Y.: The inverse methods and tactics for establishing deducibility for a calculus with functional symbols (Russian). Trudy Matem. Inst. AN SSSR **121**, 14–56 (1972). English transl. in Proc. Steklov Inst. of Mathematics 121, 11–60, AMS, Providence (1974)
12. McCune, W.: Prover9 and Mace4 (2005–2010). <https://www.cs.unm.edu/~mccune/mace4/>
13. McLaughlin, S., Pfenning, F.: Efficient intuitionistic theorem proving with the polarized inverse method. In: Schmidt, R.A. (ed.) CADE 2009. LNCS (LNAI), vol. 5663, pp. 230–244. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02959-2\\_19](https://doi.org/10.1007/978-3-642-02959-2_19)

14. Mints, G.: Resolution strategies for the intuitionistic logic. In: Mayoh, B., Tyugu, E., Penjam, J. (eds.) *Constraint Programming*. NATO ASI F, vol. 131, pp. 289–311. Springer, Heidelberg (1994). [https://doi.org/10.1007/978-3-642-85983-0\\_11](https://doi.org/10.1007/978-3-642-85983-0_11)
15. Mints, G.: Decidability of the class E by Maslov’s inverse method. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) *Fields of Logic and Computation*. LNCS, vol. 6300, pp. 529–537. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-15025-8\\_26](https://doi.org/10.1007/978-3-642-15025-8_26)
16. Orevkov, V.P.: Obratnyi metod poiska vyvoda dlia skulemovskikh predvarennykh formul ischisleniia predikativ (Russian). In: Adamenko, A.N., Kuchukov, A.M.: *Logicheskoe programmirovaniie i Visual Prolog*, appendix 4, 952–965, BHV, St. Petersburg (2003)
17. Otten, J.: Non-clausal connection-based theorem proving in intuitionistic first-order logic. In: *Proceedings of the 2nd International Workshop on Automated Reasoning in Quantified Non-Classical Logics (ARQNL/IJCAR 2016)*, CEUR Workshop Proceedings, vol. 1770, pp. 9–20 (2016). <http://ceur-ws.org/Vol-1770/>
18. Pavlov, V., Pak, V.: The inverse method application for non-classical logics. *Non-linear Phenom. Complex Syst.* **18**(2), 181–190 (2015)
19. Pavlov, V.A., Pak, V.G.: An experimental computer program for automated reasoning in intuitionistic logic using the inverse method (Russian). *SPbSPU J. Comput. Sci. Telecommun. Control Syst.* **6**(234), 70–80 (2015)
20. Raths, T., Otten, J., Kreitz, C.: The ILTP library: benchmarking automated theorem provers for intuitionistic logic. In: Beckert, B. (ed.) *TABLEAUX 2005*. LNCS (LNAI), vol. 3702, pp. 333–337. Springer, Heidelberg (2005). [https://doi.org/10.1007/11554554\\_28](https://doi.org/10.1007/11554554_28)
21. Schmitt, S., Lorigo, L., Kreitz, C., Nogin, A.: JProver: integrating connection-based theorem proving into interactive proof assistants. In: Goré, R., Leitsch, A., Nipkow, T. (eds.) *IJCAR 2001*. LNCS, vol. 2083, pp. 421–426. Springer, Heidelberg (2001). [https://doi.org/10.1007/3-540-45744-5\\_34](https://doi.org/10.1007/3-540-45744-5_34)
22. Sutcliffe, G., Suttner, C.: Evaluating general purpose automated theorem proving systems. *Artif. Intell.* **131**(1–2), 39–54 (2001)
23. Tammet, T.: A resolution theorem prover for intuitionistic logic. In: McRobbie, M.A., Slaney, J.K. (eds.) *CADE 1996*. LNCS, vol. 1104, pp. 2–16. Springer, Heidelberg (1996). [https://doi.org/10.1007/3-540-61511-3\\_65](https://doi.org/10.1007/3-540-61511-3_65)
24. The ILTP Library. Provers and Results. <http://iltp.de/results.html>
25. Voronkov, A.: Theorem proving in non-standard logics based on the inverse method. In: Kapur, D. (ed.) *CADE 1992*. LNCS, vol. 607, pp. 648–662. Springer, Heidelberg (1992). [https://doi.org/10.1007/3-540-55602-8\\_198](https://doi.org/10.1007/3-540-55602-8_198)

# Distributed In Situ Processing of Big Raster Data in the Cloud

Ramon Antonio Rodrigues Zalipynis<sup>(✉)</sup>

National Research University Higher School of Economics,  
Moscow, Russia  
arodrigues@hse.ru

**Abstract.** A raster is the primary data type in Earth science, geology, remote sensing and other fields with tremendous growth of data volumes. An array DBMS is an option to tackle big raster data processing. However, raster data are traditionally stored in files, not in databases. Command line tools have long been developed to process raster files. Most tools are feature-rich and free but optimized for a single machine. This paper proposes new techniques for distributed processing of raster data directly in diverse file formats by delegating considerable portions of work to such tools. An  $N$ -dimensional array data model is proposed to maintain independence from the files and the tools. Also, a new scheme named GROUP–APPLY–FINALLY is presented to universally express the majority of raster data processing operations and streamline their distributed execution. New approaches make it possible to provide a rich collection of raster operations at scale and outperform SciDB over  $410\times$  on average on climate reanalysis data. SciDB is the only freely available distributed array DBMS to date. Experiments were carried out on 8- and 16-node clusters in Microsoft Azure Cloud.

**Keywords:** Big raster data · Climate reanalysis  
Distributed systems · Cloud computing · SciDB · Array DBMS  
In situ · NetCDF operators

## 1 Introduction

Modern volumes of raster data are overwhelming. The most prominent examples of big raster data come from Earth remote sensing and climate modeling. For example, DigitalGlobe is the largest commercial satellite imagery provider collecting 70 TB of imagery on an average day with their constellation of six large satellites [9]. European Centre for Medium-Range Weather Forecasts (ECMWF) has alone accumulated 137.5 million files sized 52.7 PB in total [7].

The long history of file-based data storage resulted in a broad range of sophisticated raster file formats. For example, NetCDF format supports multidimensional arrays, chunking, compression, diverse data types, metadata and hierarchical namespace [10]. NetCDF has been under development since 1990 [17]

and was standardized by OGC (Open Geospatial Consortium) [13]. Decades of development and feedback resulted in numerous elaborate and quality-assured tools for processing raster files. For example, NetCDF Operators (NCO) is a set of command line tools for processing NetCDF files. NCO have been under development since 1995 [12].

In situ approach processes data directly in their native file formats. The benefits of in situ approach are widely recognized [2, 4, 5, 21, 22]. Raster file formats are in the spotlight while already existing and highly-optimized command line tools are largely ignored in this research trend: raster operations are re-implemented almost from scratch. For example, SciDB was first released in 2008 and still lacks even core raster operations like interpolation [8].

The idea of partially delegating raster data processing to existing command line tools was first presented and proved to outperform SciDB  $3\times$  to  $193\times$  in [19]. This paper extends [19] with new theory, techniques, and experimental results further showing that the tools may be a valuable component of an array DBMS.

The delegation ability is being integrated into ChronosServer [18]. Raster files in diverse file formats are distributed among cluster nodes. Section 2 presents a formalized logical data model to abstract from the files and the distributed environment: a user works with a single  $N$ -d array backed by a set of raster files on a computer cluster. The model is used to design generic distributed algorithms for in situ processing of arbitrary  $N$ -d arrays (Sect. 3).

It is formally shown that the tools are widely applicable: each algorithm has major portions of work which it delegates to a tool by directly launching this tool on respective files residing on the same cluster node. ChronosServer is responsible for proper exchange of the output files produced by the tools between cluster nodes. This enables efficient raster data processing within a single machine and provides a rich collection of raster processing operations at scale (Sect. 4).

## 2 ChronosServer

### 2.1 ChronosServer Multidimensional Array Model

In this paper, an  $N$ -dimensional array ( $N$ -d array) is the mapping  $A : D_1 \times D_2 \times \dots \times D_N \mapsto \mathbb{T}$ , where  $N > 0$ ,  $D_i = [0, l_i) \subset \mathbb{Z}$ ,  $0 < l_i$  is a finite integer, and  $\mathbb{T}$  is a standard numeric type<sup>1</sup>.  $l_i$  is said to be the *size* or *length* of  $i$ th dimension (in this paper,  $i \in [1, N] \subset \mathbb{Z}$ ). Let us denote the  $N$ -d array by

$$A\langle l_1, l_2, \dots, l_N \rangle : \mathbb{T} \quad (1)$$

By  $l_1 \times l_2 \times \dots \times l_N$  denote the *shape* of  $A$ , by  $|A|$  denote the *size* of  $A$  such that  $|A| = \prod_i l_i$ . A *cell* or *element* value of  $A$  with integer indexes  $(x_1, x_2, \dots, x_N)$  is referred to as  $A[x_1, x_2, \dots, x_N]$ , where  $x_i \in D_i$ . Each cell value of  $A$  is of type  $\mathbb{T}$ . The array may be initialized after its definition by enumerating the values of the

<sup>1</sup> To be specific about value ranges, size in bytes, precision, etc., a standard C++ type can be taken for  $\mathbb{T}$  according to ISO/IEC 14882.

cells. For example, the following defines and initializes a 2-d array of integers:  $A\langle 2, 2 \rangle : \text{int} = \{\{1, 2\}, \{3, 4\}\}$ . In this example,  $A[0, 0] = 1$ ,  $A[1, 0] = 3$ ,  $|A| = 4$ , and the shape of  $A$  is  $2 \times 2$ .

Indexes  $x_i$  are optionally mapped to specific values of  $i$ th dimension by *coordinate* arrays  $A.d_i\langle l_i \rangle : \mathbb{T}_i$ , where  $\mathbb{T}_i$  is a totally ordered set, and  $d_i[j] < d_i[j + 1]$  for all  $j \in D_i$ . In this case,  $A$  is defined as

$$A(d_1, d_2, \dots, d_N) : \mathbb{T} \quad (2)$$

A *hyperslab*  $A' \sqsubseteq A$  is an  $N$ -d subarray of  $A$ . The hyperslab  $A'$  is defined by the notation

$$A[b_1 : e_1, \dots, b_N : e_N] = A'(d'_1, \dots, d'_N) \quad (3)$$

where  $b_i, e_i \in \mathbb{Z}$ ,  $0 \leq b_i \leq e_i < l_i$ ,  $d'_i = d_i[b_i : e_i]$ ,  $|d'_i| = e_i - b_i + 1$ , and for all  $y_i \in [0, e_i - b_i]$  the following holds

$$A'[y_1, \dots, y_N] = A[y_1 + b_1, \dots, y_N + b_N] \quad (4a)$$

$$d'_i[y_i] = d_i[y_i + b_i] \quad (4b)$$

Equations (4a) and (4b) state that  $A$  and  $A'$  have a common coordinate subspace over which cell values of  $A$  and  $A'$  coincide. Note that the original dimensionality is preserved even if some  $b_i = e_i$  (in this case, “:  $e_i$ ” may be omitted in (3)).

## 2.2 ChronosServer Datasets

A *dataset*  $\mathbb{D} = (A, M, P)$  contains a *user-* or *higher-level* array  $A(d_1, \dots, d_N) : \mathbb{T}$  and the set of *system-* or *lower-level* arrays  $P = \{(A_k, B_k, E_k, M_k, \text{node}_k)\}$ , where  $A_k \sqsubseteq A$ ,  $k \in \mathbb{N}$ ,  $\text{node}_k$  is an identifier of the cluster node storing array  $A_k$ ,  $M_k$  is metadata for  $A_k$ ,  $B\langle N \rangle : \text{int} = \{b_1, b_2, \dots, b_N\}$ ,  $E\langle N \rangle : \text{int} = \{e_1, e_2, \dots, e_N\}$  such that  $A_k = A[b_1 : e_1, \dots, b_N : e_N]$ . A user-level array is never materialized and stored explicitly: operations with  $A$  are mapped to a sequence of operations with respective arrays  $A_k$ . Let us call a user-level array and a system-level array an *array* and a *subarray* respectively for short. A dataset also contains metadata  $M = \{(key, val)\}$ , where *key* is a string and *val* is a string or a number. Dataset metadata include two types of information: general dataset properties (name, description, contacts, etc.) and metadata valid for all  $p \in P$  (array data type  $\mathbb{T}$ , storage format, etc.). For example,  $M = \{(name = \text{“Wind Speed”}), (type = \text{float}), (format = \text{NetCDF})\}$ . Let us refer to an element in a tuple  $p = (A_k, B_k, \dots) \in P$  as  $p.A$  for  $A_k$ , etc.

## 2.3 ChronosServer Architecture

ChronosServer runs on a computer cluster of commodity hardware. Files of diverse raster file formats are kept intact and distributed among cluster nodes without changing their names and formats. A file is always stored entirely on a node in contrast to parallel or distributed file systems. Workers are launched at each node and are responsible for data processing. A single Gate at a dedicated

node receives client queries and coordinates workers. A file may be replicated on several workers for fault tolerance and load balancing.

The Gate stores metadata for all datasets and their subarrays. Consider a dataset  $\mathbb{D} = (A, M, P)$ . Arrays  $A.d_i$  and elements of  $\forall p \in P$  except  $p.A$  are stored on the Gate. In practice, array axes usually have coordinates such that  $A.d_i[j] = start + j \times step$ , where  $j \in [0, |A.d_i|) \subset \mathbb{N}$ ,  $start, step \in \mathbb{R}$ : only  $|A.d_i|$ ,  $start$  and  $step$  values have to be usually stored. ChronosServer array model merit is that it has been designed to be as generic as possible but allowing the establishment of 1:1 mapping of a subarray to a dataset file.

Upon startup workers connect to the Gate and receive the list of all available datasets and file naming rules. Workers scan their local filesystems to discover datasets and create  $p.M, p.B, p.E$  by parsing file names or reading file metadata. Workers transmit to the Gate all elements of  $p$  except  $p.A$ .

## 2.4 GROUP-APPLY-FINALLY (GAF)

The proposed scheme described in this subsection makes it possible to universally express the majority of raster data processing operations and organize their distributed execution in a clear and uniform manner.

Raster operations can be classified as global (involve all data), local (cell-wise), focal (cell values from a rectangular window are required to compute the new cell value), zonal (the same as focal but the area is defined by a function).

Given that subarrays may be distributed among cluster nodes, some operations cannot be completed autonomously using data on a single node. For example, consider daily data 01.01.1979, 02.01.1979, 03.01.1979 for a dataset with 6 hour time step (each subarray contains data for one day, e.g. 4 time steps for hours 00, 06, 12 and 18). The interpolation of data from 6 to 3h step for the subarray with data for 02.01.1979 requires the presence of the both subarrays for 01.01.1979 and 03.01.1979 on the same node.

The new scheme enables to perform raster operations that involve subarrays residing on several nodes: `GROUP pattern APPLY  $F_A$  COMBINE  $F_C$  FINALLY  $F_F$` , where  $F_A$ ,  $F_C$ , and  $F_F$  are sequences of raster operations. All phases except `APPLY` are optional. If present, `GROUP` phase serves as a preparation step before executing an `APPLY` phase: *pattern* defines a rule according to which subarrays are to be collocated within a single node to apply  $F_A$  on them.

For example, in the case of interpolation described above, the pattern may state that there should be at least two subarrays on the same cluster node containing data from the next and the previous days (except for the first and the last subarray in the available time interval). Satisfying *pattern* conditions may require data movement between cluster nodes. `GROUP` phase ensures that the nodes involved in the computations satisfy the grouping criterion required to enable autonomous execution of the next phases.

After a `GROUP` phase, an `APPLY` phase is launched on each involved node. Since some nodes may have files from disjoint temporal or spatial intervals, their intermediate results may be combined on the same node to reduce network traffic

with  $F_C$  on COMBINE phase if possible. All results (output subarrays after  $F_A$  or  $F_C$ ) are gathered on a single node and  $F_F$  is applied to obtain the final result.

The GROUP phase and subarray gathering for the FINALLY phase are realized using STREAM [node] *dataset*<sub>1</sub> and CONSUME [count] *dataset*<sub>2</sub> phases. STREAM phase sends all subarrays of *dataset*<sub>1</sub> to the node with identifier *node*. CONSUME accepts *count* subarrays from other nodes into the dataset named *dataset*<sub>2</sub>.

Unlike existing schemes, the proposed distributed execution scheme takes into account peculiarities inherent to raster operations and the file-based storage model. For example, respective grouping pattern must be specified to guarantee the possibility of computing intermediate results autonomously on each node.

## 2.5 New Delegation Approach

The ChronosServer array data model has two levels of abstraction. This leads to two levels of array processing commands: user-level and system-level. The latter commands are expressed in terms of the GAF scheme by providing *pattern*,  $F_A$ ,  $F_C$ , and  $F_F$  what may be difficult for a ChronosServer user. System-level processing explicitly takes place on subarrays. A user-level command is defined on user-level  $N$ -d arrays and is mapped to a predefined GAF structure.

ChronosServer users operate with a familiar command line syntax: the names of ChronosServer user-level commands coincide with the names of existing command line tools. Command options have the same meaning and names as for the tool but without options related to file names or paths in order to abstract from the notion of a file.

The syntax of a ChronosServer command is the same as launching a tool from a command line. Input and output dataset names must be specified instead of input and output file names. For example, the command below is analogous to `ncra` (NetCDF Record Average) utility from NCO package (STREAM and CONSUME phases are not shown). The command calculates the sum of all array cells along the first dimension (the input dataset description is in Sect. 4).

```

user-level:  ncra -D 2 -y ttl r2.wind.u10m.u r2.wind.u10mtotal
system-level: APPLY ncra -D 2 -y ttl r2.wind.u10m.u_key:all $u.total
              FINALLY ncra -D 4 -y ttl $u.total:all r2.wind.u10mtotal

```

The proposed theoretical framework makes it straightforward to partially delegate array processing to an external command line tool. For some functions from  $F_A$ ,  $F_C$ , and  $F_F$  an equivalent functionality can be achieved by running a command line tool on a subarray (file).

The input dataset is “r2.wind.u10m.u” (due to the large number of datasets their namespace is hierarchical, the levels of hierarchy are separated by dots). The output dataset name is “r2.wind.u10mtotal”. Intermediate datasets are prefixed with “\$” and are reused in subsequent commands in  $F_A$ ,  $F_C$ , or  $F_F$ . Intermediate datasets are not registered as regular datasets on the Gate to save time and network traffic. Dataset `r2.wind.u10m.u_key` is created by the Gate and contains the subset of subarrays, Sect. 3.1. Several system-level commands are created to perform the aggregation of a dataset, Sect. 3.1.



Dataset quantiles are specified after a colon “:”. They are designed to fully leverage the power of the tools. For example, `ncra` can take several files as input. It is reasonable not to launch `ncra` on each file and then combine the result into a single file on `COMBINE` phase but to feed all locally available files to `ncra` on `APPLY` phase to reduce the computation and I/O time. This is exactly the behavior triggered by the “all” quantile.

### 3 Array Operations

#### 3.1 Aggregation

The aggregate of an  $N$ -d array  $A(d_1, d_2, \dots, d_N) : \mathbb{T}$  over axis  $d_1$  is the  $(N - 1)$ -d array  $A_{aggr}(d_2, \dots, d_N) : \mathbb{T}$  such that  $A_{aggr}[x_2, \dots, x_N] = f_{aggr}(cells(A[0 : |d_1| - 1, x_2, \dots, x_N]))$ , where  $x_2, \dots, x_N$  are valid integer indexes,  $f_{aggr} : T \mapsto w$  is an aggregation function,  $T$  is a multiset of values from  $\mathbb{T}$ ,  $w \in \mathbb{T}$ ,  $cells : A' \mapsto T$  is the multiset of all cell values of an array  $A' \subseteq A$ .

Algorithm 1 assumes that a user-level array is split onto subarrays by hyperplanes, Fig. 1. This is a usual case in practice.

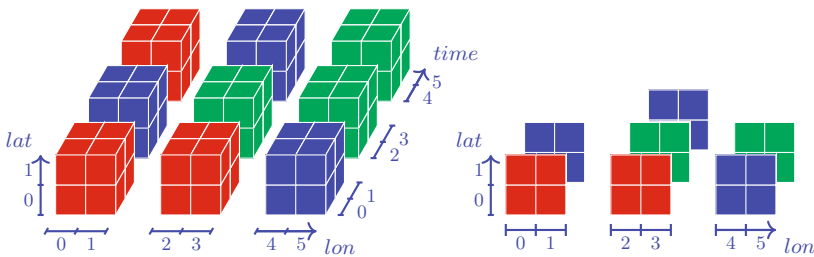


Fig. 1. Array aggregation.

Figure 1 illustrates Algorithm 1 which is executed on the Gate. Lines prefixed by `worker(node)` contain messages sent to the worker  $N^{\circ}node$ . The messages contain instructions for GAF phases. Several such messages can be packed into a single network message. Subarrays with the same color reside on the same node, workers in parallel aggregate all subarrays (the left of Fig. 1) that contribute to the same 2-d key over the first axis to obtain interim 2-d aggregates (the right of Fig. 1). The 2-d arrays having the same 2-d key are gathered on one of the nodes which calculates the final result for a particular 2-d key (not shown). The mapping  $\mu : key \mapsto id$  returns the worker ID to which the partial aggregates must be sent. For example, red and blue 2-d arrays ( $[0 : 1, 0 : 1]$ ) at the right of Fig. 1 are intermediate aggregates residing on two different nodes. The nodes will send their subarrays to the node  $\mu(0, 0)$  to calculate the final result.

---

**Algorithm 1.** Distributed In Situ Array Aggregation

---

```

Input:  $P, f_{aggr}, \mu$    Output:  $P_{aggr}$    Require:  $N > 1$ 
1: for each  $key \in \{p.key[1:N-1] : p \in P\}$  do
2:    $\mathbb{C} \leftarrow \{p : p.key[1:N-1] = key \wedge p \in P\}$  and  $Nodes \leftarrow \{c.node : c \in C\}$ 
3:   for each  $node \in Nodes$  do
4:      $\mathbb{P} \leftarrow \{p : p.node = node \wedge p \in \mathbb{C}\}$ 
5:     worker( $node$ ): APPLY ncra -y  $f_{aggr}$   $\mathbb{P}$ :all $interim ▷ Delegation
6:     if  $node \neq \mu(key)$  then worker( $node$ ): STREAM[ $\mu(key)$ ] $interim
7:     worker( $\mu(key)$ ): CONSUME[ $[Nodes \setminus \{\mu(key)\}]$ ] $interim
8:     worker( $\mu(key)$ ): FINALLY ncra -y  $f_{aggr}$  $interim:all  $P_{aggr}$  ▷ Delegation

```

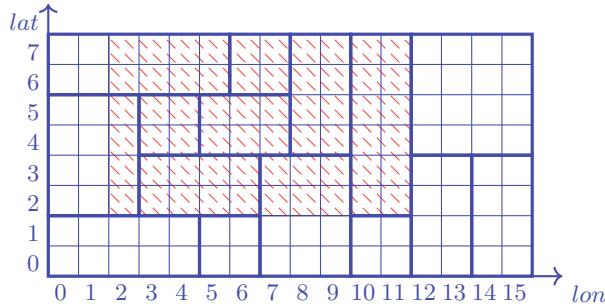
---

Algorithm 1 is for  $f_{aggr} \in \{max, min, sum\}$ . Calculation of *avg* is reduced to calculating the *sum* and dividing each cell of the resulting array on  $|A.d_1|$ .

Lines 5 and 8 of Algorithm 1 are highlighted in gray to accent the work being delegated to an external tool.

### 3.2 Hyperslabbing

*Hyperslabbing* is an extraction of a hyperslab from an array, Eq. (3). Consider a 2-d array  $A(lat, lon)$ , Fig. 2. Array  $A$  has shape  $8 \times 16$  and consists of 15 subarrays that are separated from each other by thick lines and may reside on different cluster nodes. The hatched area marks the hyperslab  $A' = A[2:7, 2:11]$ .



**Fig. 2.** Array hyperslabbing.

Hyperslabbing an array is reduced to hyperslabbing the respective subarrays as follows. Some subarrays do not participate in the hyperslabbing: almost 40% of them do not overlap with  $A'$  and can be filtered out beforehand (e.g.,  $A[0:1, 0:4]$ ). Also, almost 40% of the subarrays are entirely inside  $A'$  and must migrate to the resulting dataset as is (e.g.,  $A[4:5, 5:7]$ ). It is possible to avoid copying these subarrays and store only references to them from the new dataset since datasets are immutable in ChronosServer.

It is necessary to hyperslab only 3 subarrays to complete the operation. Since any subarray is entirely located on a machine in a single file, hyperslabbing a

subarray is delegated to a command line tool. Almost every tool supports raster file subsetting. However, most tools work on a single machine. ChronosServer scales them out by orchestrating their massively parallel execution.

**Algorithm 2.** Distributed in situ array hyperslabbing with delegation to an external command line tool (procedure HYPERSLAB is executed on the Gate).

```

1: procedure HYPERSLAB( $\mathbb{D}, b_1, \dots, b_N, e_1, \dots, e_N$ )  $\triangleright b_i, e_i : \text{eq. (3)}$ 
2:    $P' \leftarrow \{\}$   $\triangleright \text{require } 0 \leq b_i \leq e_i < |A.d_i| \text{ for each } i$ 
3:    $A'.d'_i \leftarrow A.d_i[b_i : e_i]$ 
4:    $m'_i \leftarrow A'.d'_i$ 
5:    $l'_i \leftarrow |m'_i| - 1$ 
6:   for each  $p \in P$  do
7:      $m_i \leftarrow p.A.d_i$ 
8:      $l_i \leftarrow |m_i| - 1$ 
9:      $s_i \leftarrow \max(m_i[0], m'_i[0])$ 
10:     $v_i \leftarrow \min(m_i[l_i], m'_i[l'_i])$ 
11:    if  $s_i \leq v_i$  for all  $i$  then  $\triangleright p.A$  and  $A'$  overlap
12:      if  $s_i = m_i[0] \wedge v_i = m_i[l_i]$  for all  $i$  then  $\triangleright p.A \sqsubseteq A'$ 
13:         $P' \leftarrow P' \cup \{p\}$ 
14:        WORKER link  $p$  to new dataset  $\mathbb{D}'$   $\triangleright \text{execute on worker}$ 
15:      else  $\triangleright \text{hyperslab } p.A$ 
16:        find  $b'_i$  and  $e'_i : m_i[b'_i] = s_i \wedge m_i[e'_i] = v_i$ 
17:        WORKER  $p'.A = p.A[b'_1 : e'_1, \dots, b'_N : e'_N]$   $\triangleright \text{DELEGATION}$ 
18:         $P' \leftarrow P' \cup \{p'\}$ 
19:  return  $\mathbb{D}' = (A', M, P')$   $\triangleright A'(d'_1, \dots, d'_N) = A[b_1 : e_1, \dots, b_N : e_N], \text{eq. (3)}$ 

```

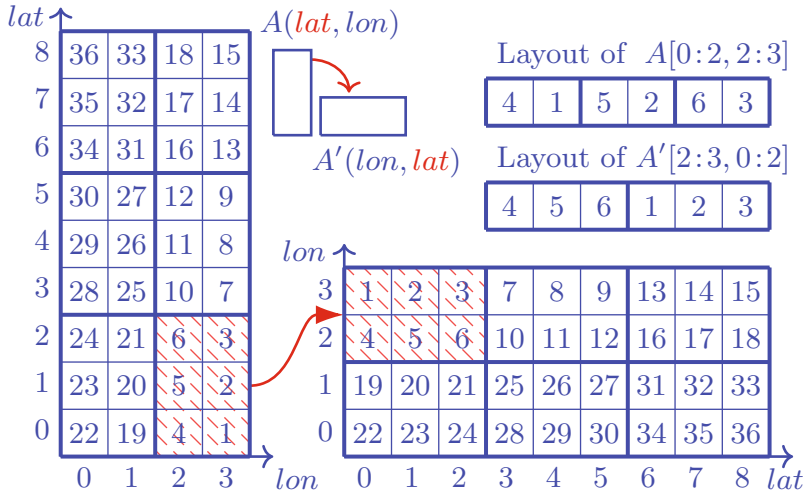
Algorithm 2 outlines the hyperslabbing algorithm. Line 17 is highlighted in light gray to accent the work which is possible to delegate to an external tool: ncks (NCO) for NetCDF format.

### 3.3 Reshaping

The *reshaping* operation  $\Psi : A, \pi \mapsto A'$  takes as input an  $N$ -d array  $A(d_1, \dots, d_N) : \mathbb{T}$  and the permutation mapping  $\pi : i \mapsto j$ , where  $i, j \in [1, N] \subset \mathbb{N}$ ,  $\pi(i) \neq \pi(j)$  for  $i \neq j$ , and  $\bigcup_i \{\pi(i)\} = [1, N]$ . The reshaping operation outputs the  $N$ -d array  $A'(d_{\pi(1)}, \dots, d_{\pi(N)}) : \mathbb{T}$  such that  $A[x_1, \dots, x_N] = A'[x_{\pi(1)}, \dots, x_{\pi(N)}]$ , where  $x_i \in [0, |d_i|) \subset \mathbb{N}$  for all  $i$ .

Reshaping an array is reduced to the independent reshaping of its subarrays. Figure 3 depicts array  $A(lat, lon) : \text{int}$  (shape  $9 \times 4$ ) and its reshaped version  $A'(lon, lat) : \text{int}$  (shape  $4 \times 9$ ), the dimensions of  $A$  and  $A'$  are swapped. Subarrays of  $A$  and  $A'$  are separated by thick lines.

Hatched areas within  $A$  and  $A'$  correspond to subarrays  $A[0 : 2, 2 : 3]$  and  $A'[2 : 3, 0 : 2]$  respectively. Subarray  $A'[2 : 3, 0 : 2]$  is the reshaped version of subarray  $A[0 : 2, 2 : 3]$ . Subarray  $A'[b_{\pi(1)} : e_{\pi(1)}, \dots, b_{\pi(N)} : e_{\pi(N)}]$  is the reshaped version of subarray  $A[b_1 : e_1, \dots, b_N : e_N]$ . Figure 3 clearly shows that no exchange of elements between distinct subarrays of  $A$  does not occur during reshaping. In order to reshape an arbitrary  $N$ -d array, all its subarrays must be reshaped independently of each other.



**Fig. 3.** Array reshaping.

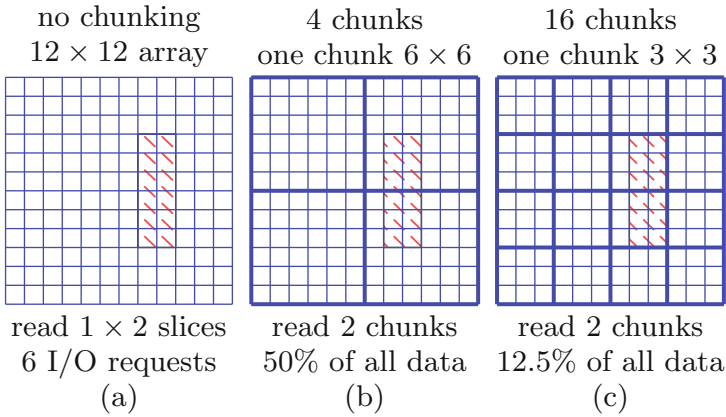
Array reshaping re-organizes the storage layout of array cells according to the given order of array dimensions. An  $N$ -d array has a linear storage layout in the memory. Usually the last dimension of an  $N$ -d array varies the fastest (i.e. elements along the  $N$ th dimension are sequential in the memory). The storage layouts of subarrays  $A[0:2, 2:3]$  and  $A'[2:3, 0:2]$  are depicted on Fig. 3 as two strips (1-d arrays). A cell with index  $(x_1, x_2)$  precedes the cell  $(x_1, x_2 + 1)$ . Thick lines on the strips separate elements with distinct  $x_1$  indexes.

Reshaping is used to achieve the fastest possible performance of reading array data along a certain dimension. Consider a 3-d array  $A(\text{time}, \text{lat}, \text{lon})$ . The  $\text{lon}$  and  $\text{time}$  dimensions vary the fastest and the slowest respectively. It is required to issue  $|A.d_1|$  I/O requests to read the time series  $A[0 : |A.d_1| - 1, x_2, x_3]$  if  $A$  is not chunked (Sect. 3.4). However, only a single I/O request will suffice to read the time series from the reshaped array  $A(\text{lat}, \text{lon}, \text{time})$ . It is possible for chunking to make up some of the performance difference. However, no chunking scheme will be as fast as just re-ordering the data. This is especially important in the Cloud which usually limits IOPS (I/O operations per second). Reshaping is delegated to `ncpdq` tool (NCO) for NetCDF file format.

### 3.4 Chunking

Chunking is the process of partitioning original array into a set of smaller sub-arrays called chunks. Chunks are autonomous, possibly compressed subarrays (hyperslabs) with contiguous storage layout. Given chunk shape  $c_1 \times \dots \times c_N$  and an  $N$ -d array  $A(d_1, \dots, d_N) : \mathbb{T}$ , the *exact chunking* operation reorganizes cells in array  $A$  such that all cells of  $A$  with coordinates  $(x_1, \dots, x_N)$  and  $(y_1, \dots, y_N)$  belong to the same chunk if  $x_i \text{ div } c_i = y_i \text{ div } c_i$  for all  $i$ .

A chunk is usually read/written completely from/to disk in one request to a storage subsystem. Chunking is one of the classical approaches to significantly accelerate disk I/O when only a portion of raster is read. Unlike reshaping (Sect. 3.3), chunking does not alter array metadata: a cell index remains the same after chunking. Chunking is often used when there is no space to store the reshaped version of an array or when the most frequent array access patterns are not known in advance.



**Fig. 4.** Array chunking.

Consider reading a  $6 \times 2$  slice from a 2-d array, Fig. 4. For a row-major storage layout, two vertically adjacent cells are located far apart each other. A possible solution is to read 6 portions sized  $1 \times 2$  which requires 6 I/O requests and disk seeks, Fig. 4a. For a compressed array, much larger part of it might be required to be read and uncompressed before getting the requested portion.

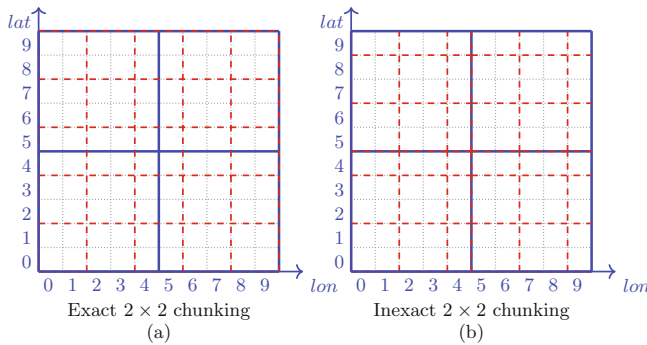
In contrast, only chunks containing required data are read from disk from a chunked array. However, inappropriate chunk shape may result in a large I/O overhead, Fig. 4b. Good chunk shape allows to reduce communication with storage layer, disk seeks and I/O volume, Fig. 4c. This is especially prominent in a Cloud where IOPS (I/O operations per second) are always limited and billed for. Since many raster operations are mostly I/O bound [23], chunk shape is one of the crucial performance parameters for a dataset [6].

Chunk shape depends on data characteristics and workload. Optimal chunk shape usually does not exist for all access patterns. It is also difficult to guess a good chunk shape a priori: chunk shape is often tuned experimentally. An array DBMS must be capable to quickly alter chunk shape in order to support experimentation as well as to adapt to dynamic workloads.

The exact chunking of an array may lead to data movement between files and cluster nodes. However, in practice the condition  $c_i \ll |A.d_i|$  usually holds. This translates to  $c_i \ll |p.A.d_i|$  for  $\forall p \in P$  (in practice, raster data are already

shipped in wisely cut files satisfying this condition). For example, in climate modeling it is common to split a time series with hourly time step into files storing yearly or monthly data.

A practical approach is to do inexact user-level array chunking and exact independent chunking of its subarrays. More chunks will smaller shapes than the given one will appear, Fig. 5. However, the fraction of such small chunks will be negligible and they will not influence the I/O performance significantly. Note that if  $|A.d_i| \bmod c_i \neq 0$ , then even the exact chunking of a user-level array is not possible leading to a certain amount of chunks with smaller shapes.



**Fig. 5.** Inexact array chunking.

In practice, inexact chunking is even more desirable in many cases: it is much faster and more consistent than the exact chunking. Recall that files under ChronosServer control are directly accessible by a user and any other software. Consider the climate modeling example given above. In this case, it is inconsistent to have a perfectly chunked file named “2015.\*” and supposed to store data for year 2015 but with extra grids from the next and/or previous years.

ChronosServer does not move data between cluster nodes to alter chunk shape of a user-level array; every system-level array is chunked autonomously. ChronosServer delegates chunking to `ncks` (NCO).

## 4 Performance Evaluation

### 4.1 Experimental Setup

Microsoft Azure Cloud was used for the experiments. Azure cluster creation, scaling up and down with given network parameters, number of virtual machines, etc. was fully automated using Java Azure SDK. We used the latest SciDB version 16.9 released in November, 2016. The latest version of Ubuntu Linux on which SciDB 16.9 runs is 14.04 LTS. Standard D2 v2 virtual machines were used with 2 CPU cores (Intel Xeon E5-2673 v3 (Haswell) 2.4 GHz), 7 GB RAM,

100 GB local SSD drive (4 virtual data disks), max  $4 \times 500$  IOPS. SciDB cluster deployment is extremely labor-intensive. SciDB must be compiled from scratch directly on a computer cluster in order to deploy it on a cluster.

Although Azure states the disks to be SSD, after the creation of such a disk Azure displays it to be a standard HDD disk backed by a magnetic drive. This leads to a large spread of I/O latencies among cluster nodes, makes some nodes in the cluster much slower than the others, and explains larger runtimes for 16 nodes than for 8 nodes for ChronosServer aggregation queries, Table 1. Possibly the ChronosServer aggregation runtime is comparable to the I/O overhead of the HDD disks. This can be concluded from the hot runtimes, Table 1.

Eastward (U-wind) wind speed at 10 meters above surface from NCEP/DOE AMIP-II Reanalysis (R2) between 1979–1994 (16 years) was used for experiments [11]. These are 6-hourly forecast data (4-times daily values at 00:00, 06:00, 12:00, and 18:00). Data are 3-dimensional on 94 latitudes  $\times$  192 longitudes Gaussian grid in NetCDF3 format. This comprised only 804 MB in NetCDF3 format since it is impossible to import large data volumes into SciDB in a rea-

**Table 1.** Results of performance evaluation

Operation	Nodes	Execution Time, seconds			SciDB/ Chronos	
		ChronosServer		SciDB	Cold	Hot
		Cold	Hot			
Average	8	2.82	1.16	59.06	20.94	50.91
	16	3.5	0.53	30.14	8.61	56.87
Maximum	8	1.21	0.83	37.88	31.30	45.64
	16	1.94	0.53	17.58	9.06	33.17
Minimum	8	1.11	0.84	33.88	30.52	40.33
	16	1.42	0.49	17.46	12.30	35.63
Chunk $10 \times 10 \times 8$	16	2.15	1.96	3205.94	1491.13	1635.68
Chunk $100 \times 20 \times 16$	8	1.42	1.29	314.88	221.75	244.09
	16	1.02	0.63	211.56	207.41	335.81
Reshape <sup>1</sup>	8	9.51	5.64	345.55	36.33	61.27
	16	2.93	2.86	196.85	67.18	68.83
Hyperslab original <sup>2</sup>	8	2.64	0.24	263.59	99.84	1098.29
	16	1.40	0.22	116.80	83.43	530.91
Hyperslab chunked <sup>3</sup>	8	6.11	0.39	3.46	0.57	8.87
	16	2.55	0.27	1.95	0.76	7.22

<sup>1</sup>  $A(\text{time}, \text{lat}, \text{lon}) \rightarrow A(\text{lon}, \text{lat}, \text{time})$

<sup>2</sup>  $A[0 : 23360 - 1, 0, 0]$  from  $A(\text{time}, \text{lat}, \text{lon})$

<sup>3</sup>  $A[0 : 23360 - 1, 0, 0]$  from chunked  $100 \times 20 \times 16$

sonable time frame [19]. However, small test data volume and single machine turned out to be sufficient for representative results (Table 1).

We have written a Java program that converts NetCDF files to CSV files to feed the latter to SciDB. To date, this is the only way to import an external NetCDF file into SciDB 16.9. The resulting shape of the wind speed array was  $23360 \times 94 \times 192$  (*time, lat, lon*). Import time of the 16 years of data into SciDB took about 27 h on a powerful local machine in order not to waste the Cloud time. The resulting SciDB array was exported into the file of proprietary SciDB binary format and copied into the Cloud when needed (SciDB imports data from its own format much faster: 148.32 s on 16 nodes).

SciDB is mostly written on C++, parameters used: 0 redundancy, 2 instances per machine, other settings are default. ChronosServer has 100% Java code, ran one worker per node, Oracle JDK 1.8.0\_111 64 bit, max heap size 978 MB (-Xmx). The cloud was running only ChronosServer workers. Gate was running on a separate VPS (Virtual Private Server) within another datacenter to simplify the overall deployment procedure. Workers connected to the gate via the Internet. We used NCO tools available from the standard Ubuntu 14.04 repository: v4.4.2, last modified 2014/02/17.

We have evaluated cold and hot query runs (a query is executed for the first and for the second time respectively on the same data). Every runtime reported is the average of 3 runtimes of the same query. Respective OS commands were issued to free `pagecache`, `dentries` and `inodes` each time before executing a cold query to prevent data caching at various OS levels. ChronosServer benefits from native OS caching and is much faster during hot runs when the same query is executed the second time on the same data. This is particularly useful for continues experiments with the same data. This type of experiments occurs quite often (e.g., tuning certain parameters, refer to Sect. 3.4 for an example). There is no significant runtime difference between cold and hot SciDB runs.

The result of aggregate queries presented in Table 1 is a single  $94 \times 192$  grid. NetCDF files were evenly distributed among cluster nodes. SciDB also evenly distributes its chunks (this can be checked by `summarize` SciDB plugin). SciDB array had chunk shape  $1 \times 94 \times 192$  which is similar to NetCDF data. Experiments for chunking the array to  $10 \times 10 \times 8$  is reported only for the 16-node cluster due to its high runtime.

## 5 Related Work

Four modern raster data management trends are relevant to this paper: industrial raster data models, formal array models and algebras, in situ data processing algorithms, and array DBMS. A good survey on the algorithms is in [4]. A recent survey of existing array DBMS and similar systems is in [19]. It is worth mentioning SciDB [6], Oracle Spatial [14], ArcGIS IS [1], RasDaMan [2], MonetDB SciQL [24], Intel TileDB [15, 20], and PostGIS [16].

The most well-known array models and algebras targeted at dense multidimensional, general-purpose arrays are Array Algebra, AML (Array Manipulation



Language), AQL (Array Query Language) and RAM. All of them can be mapped to Array Algebra [3]. SciDB does not have a formal description of its data model. It neither allows array dimensions to be of temporal or spatial types making it difficult or sometimes impossible to process many real-world datasets.

Three industry standard data models are most widely used to abstract from raster file formats: Unidata CDM, GDAL Data Model and ISO 19123 which are mappable to each other [10]. Each data model has resulted from decades of considerable practical experience. Unlike CDM, GDAL or ISO, our data model is formalized and a subarray (not a dataset) representation resembles CDM to (i) leverage industrial experience, (ii) provide a rich set of data types (Gaussian, irregular grids, meshes, etc.), (iii) make the model mapping to a raster file format clear but still independent from a format. Most popular command line tools are readily compatible with our model since they rely on CDM, GDAL or ISO.

## 6 Conclusions

ChronosServer is gradually becoming a file based, distributed array DBMS. It delegates some raster data processing work to feature-rich and highly optimized command line tools. This makes it run much faster than SciDB. Also, the ChronosServer formal array model maintains a high level of independence from the underlying raster file formats and the tools. The tools are shown to be widely applicable: in every algorithm designed for general-purpose  $N$ -d array processing there is always a significant portion of work for the tools.

Future work includes adding ACID guarantees and fault-tolerance. This may be easier than in a traditional DBMS since ChronoServer datasets are read-only.

**Acknowledgments.** This work was partially supported by Russian Foundation for Basic Research (grant No. 16-37-00416).

## References

1. ArcGIS for server — Image Extension. <http://www.esri.com/software/arcgis/arcgisserver/extensions/image-extension>
2. Baumann, P., Dumitru, A.M., Merticariu, V.: The array database that is not a database: file based array query answering in rasdaman. In: Nascimento, M.A., Sellis, T., Cheng, R., Sander, J., Zheng, Y., Kriegel, H.-P., Renz, M., Sengstock, C. (eds.) SSTD 2013. LNCS, vol. 8098, pp. 478–483. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-40235-7\\_32](https://doi.org/10.1007/978-3-642-40235-7_32)
3. Baumann, P., Holsten, S.: A comparative analysis of array models for databases. *Int. J. Database Theory Appl.* **5**(1), 89–120 (2012)
4. Blanas, S., et al.: Parallel data analysis directly on scientific file formats. In: ACM SIGMOD (2014)
5. Buck, J., et al.: SciHadoop: array-based query processing in Hadoop. In: SC (2011)
6. Cudre-Mauroux, P., et al.: A demonstration of SciDB: a science-oriented DBMS. *Proc. VLDB Endowment* **2**(2), 1534–1537 (2009)

7. Grawinkel, M., et al.: Analysis of the ECMWF storage landscape. In: 13th USENIX Conference on File and Storage Technologies, p. 83 (2015)
8. Interpolation - SciDB forum. <http://forum.paradigm4.com/t/interpolation/1283>
9. Digitalglobe's maps API. <https://www.mapbox.com/blog/digitalglobe-maps-api/>
10. Nativi, S., Caron, J., Domenico, B., Bigagli, L.: Unidata's common data model mapping to the ISO 19123 data model. *Earth Sci. Inform.* **1**, 59–78 (2008)
11. NCEP-DOE AMIP-II Reanalysis. <http://www.esrl.noaa.gov/psd/data/gridded/data.ncep.reanalysis2.html>
12. NCO homepage. <http://nco.sourceforge.net/>
13. OGC netcdf. <http://www.opengeospatial.org/standards/netcdf>
14. Oracle spatial and graph. <http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/index.html>
15. Papadopoulos, S., et al.: The TileDB array data storage manager. *Proc. VLDB Endowment* **10**, 349–360 (2016)
16. PostGIS raster data management. [http://postgis.net/docs/manual-2.2/using-raster\\_dataman.html](http://postgis.net/docs/manual-2.2/using-raster_dataman.html)
17. Rew, R., Davis, G.: NetCDF: an interface for scientific data access. *IEEE Comput. Graphics Appl.* **10**(4), 76–82 (1990)
18. Rodrigues Zalipynis, R.A.: ChronosServer: real-time access to “native” multi-terabyte retrospective data warehouse by thousands of concurrent clients. In: *Informatics, Cybernetics and Computer Engineering*, vol.14, no. 188, pp. 151–161 (2011)
19. Rodrigues Zalipynis, R.A.: ChronosServer: fast in situ processing of large multidimensional arrays with command line tools. In: Voevodin, V., Sobolev, S. (eds.) *RuSCDays 2016*. CCIS, vol. 687, pp. 27–40. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-55669-7\\_3](https://doi.org/10.1007/978-3-319-55669-7_3)
20. TileDB. <http://istc-bigdata.org/tiledb/index.html>
21. Wang, Y., Jiang, W., Agrawal, G.: SciMATE: a novel MapReduce-like framework for multiple scientific data formats. In: *CCGRID*, pp. 443–450 (2012)
22. Wang, Y., et al.: SAGA: array storage as a DB with support for structural aggregations. In: *SSDBM 2014*
23. Zender, C., et al.: Scaling properties of common statistical operators for gridded datasets. *Intl. J. High Perf. Comp. Appl.* **21**(4), 458–496 (2007)
24. Zhang, Y., et al.: SciQL: bridging the gap between science and relational DBMS. In: *IDEAS* (2011)

# Statistical Approach to Increase Source Code Completion Accuracy

Valeriy Savchenko<sup>1</sup>(✉) and Alexander Volkov<sup>2</sup>(✉)

<sup>1</sup> Institute for System Programming of the Russian Academy of Sciences,  
25, Alexander Solzhenitsyn street, Moscow 109004, Russian Federation  
vsavchenko@ispras.ru

<sup>2</sup> Lomonosov Moscow State University, GSP-1, Leninskie Gory,  
Moscow 119991, Russian Federation  
volkovas\_174@icloud.com

**Abstract.** Code completion is an essential feature in every IDE’s toolbox, boosting a developer’s productivity and significantly reducing time spent on code exploration. In this paper, we introduce the extension of a typical code completion system. At each point, we construct a list of all possible functions, which are then sorted according to our probabilistic model. We draw our inspiration from natural language processing (NLP). As the foundation, we select the N-gram model, which works on top of abstract syntax tree (AST) nodes. Since our approach is not bound to any other analyses, our model is language-agnostic, and thus, can be applied to any programming language. Experiments on several well-known open source projects show that the described method is sound. It has an execution time comparable to naïve approaches and achieves much more accurate results.

**Keywords:** Code completion · Statistics · Language modeling

## 1 Introduction

To deliver code faster and more efficiently, many software developers rely on the rich functionalities provided by different IDEs. According to the Stack Overflow Developer Survey [1], full-stack developers tend to prefer Development Environments over text editors. Studies [2, 17] show that code completion is by far the most popular IDE feature. It helps programmers to type faster, avoid misspellings and explore code without constant switching to documentation and back. Code completion is used three times more frequently than copy-paste functionality [2]. Thus code completion and the level of its performance have a drastic effect on a developer’s productivity.

Several approaches to improve code completion have been proposed in the literature. Some works extend the basic idea of code completion to improve usability. For example, the abbreviation completion algorithm [8] suggests completions for abbreviated input, while active code completion [18] suggests not

only functions, but colors and regular expressions. However, the main area of research focuses on increasing code completion precision. To achieve this, authors have proposed methods using different sources of additional information: type hierarchies and function popularity [10], program change history [20], an n-gram language model to predict the next token [9]. To improve code completion as a code exploration tool a few approaches have also been suggested. Within a database of API usage examples, either structural similarities between the user’s code and examples can be captured to provide better candidates [3], or alternatively the database can be used to train a language model to fill API calls in the given snippet [19].

In this paper, we propose a method of increasing code completion precision without any source of information outside a user’s project. Our method represents a fusion of previous ideas. With the goal of improving a user’s experience of working with her own code, and not an external API.

Our approach includes statistical language modeling that has been effectively used in several areas of NLP including sentiment analysis, speech recognition and machine translation. Studies [9, 19] show that it can also be applied to programming code. We collect data on a user’s project and construct a smoothed n-gram model. Then the model is built on top of AST nodes, thus providing the probability of a certain function call to appear in the current sub-tree. Finally, we sort completion candidates by their probabilities.

## 2 Language Model

In NLP, *language modeling* estimates the probability of the next word in a sequence of words [16]. This sequence can be of any length, whether it’s a sentence, a paper, or even a novel. A sequence of words is defined as an ordered set of words  $w_i$ , where each  $w_i$  is an element of dictionary  $\mathcal{D}$ , which is the set of all words in a particular language. If  $\{w_1, w_2, \dots, w_{m-1}\}$  (for any arbitrary  $m$ ) is a sequence from  $\mathcal{D}^*$ , then for each  $w \in \mathcal{D}$ , there is a probability of  $w$  to be  $w_m$ , i.e. the next word in the sequence.

$$P(w = w_m | w_{m-1}, w_{m-2}, \dots, w_2, w_1) \tag{1}$$

Using the formula for conditional probability, we get:

$$P(w | w_{m-1}, w_{m-2}, \dots, w_2, w_1) = \frac{P(w, w_{m-1}, w_{m-2}, \dots, w_2, w_1)}{P(w_{m-1}, w_{m-2}, \dots, w_2, w_1)} \tag{2}$$

In practice, it’s impossible to estimate both  $P(w, w_{m-1}, w_{m-2}, \dots, w_2, w_1)$  and  $P(w_{m-1}, w_{m-2}, \dots, w_2, w_1)$  for every possible length of a sequence because in order to get a proper estimation, every single possible sequence should be counted. Considering the fact that  $|\mathcal{D}^*| = |\mathbb{N}|$ , this task is equal to counting all the elements in  $\mathbb{N}$ . Since it’s impossible to do this, a major challenge for NLP is to find a way to estimate (1).

### 2.1 N-gram Model

One of the most widespread approaches to this problem is to make the *Markov assumption*, which states that the target probability depends only on the  $n - 1$  last elements of the sequence, i.e.

$$P(w|w_{m-1}, w_{m-2}, \dots, w_2, w_1) = P(w|w_{m-1}, \dots, w_{m-n+1}) \tag{3}$$

Let's denote  $\gamma_n(m)$  as the  $n - 1$  previous words for the  $m$ -th word, i.e.  $\gamma_n(m) = w_{m-1}, \dots, w_{m-n+1}$ . Then the target probability (3) takes the following form:

$$P(w|w_{m-1}, w_{m-2}, \dots, w_2, w_1) = P(w|\gamma_n(m)) \tag{4}$$

Joining (2) and (4), we get a formula to estimate the target probability:

$$\hat{P}(w|\gamma_n(m)) = \frac{\hat{P}(\gamma_n(m), w)}{\hat{P}(\gamma_n(m))} \tag{5}$$

To estimate probabilities in the right-hand side of (5), we can use relative frequencies:

$$\hat{P}(\gamma_n(m), w) = \frac{1}{|\mathcal{S}_n|} \sum_{s \in \mathcal{S}_n} \mathbf{1}_{s=\gamma_n(m), w} \tag{6}$$

$$\hat{P}(\gamma_n(m)) = \frac{1}{|\mathcal{S}_{n-1}|} \sum_{s \in \mathcal{S}_{n-1}} \mathbf{1}_{s=\gamma_n(m)} \tag{7}$$

where  $\mathcal{S}_n$  is a set of all sequences of the length  $n$  that occur in a sample. Using (6) and (7) in (5), we get the resulting formula for estimation:

$$\hat{P}(w|\gamma_n(m)) = \frac{|\mathcal{S}_{n-1}|}{|\mathcal{S}_n|} \frac{\sum_{s \in \mathcal{S}_n} \mathbf{1}_{s=\gamma_n(m), w}}{\sum_{s \in \mathcal{S}_{n-1}} \mathbf{1}_{s=\gamma_n(m)}} \tag{8}$$

Considering the fact that for a given sample,  $\frac{|\mathcal{S}_{n-1}|}{|\mathcal{S}_n|}$  is a constant, (8) is usually simplified to:

$$\hat{P}(w|\gamma_n(m)) \simeq \frac{\sum_{s \in \mathcal{S}_n} \mathbf{1}_{s=\gamma_n(m), w}}{\sum_{s \in \mathcal{S}_{n-1}} \mathbf{1}_{s=\gamma_n(m)}} \tag{9}$$

The resulting estimate (9) is referred to as the *maximum likelihood* (ML) estimate.

### 3 Programming Language Model

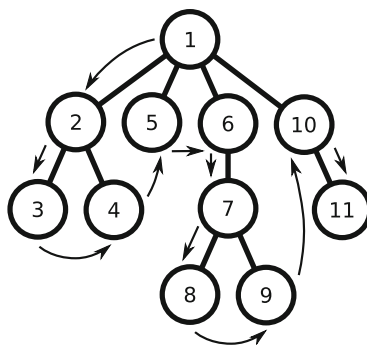
Our central hypothesis is that in various contexts, functions have different probabilities to be called, and furthermore, these probabilities vary enough that they can be exploited for better completions. Figure 1 shows one possible scenario. `std::vector`'s most callable functions are `push_back` and `operator[]`, except in the context of a `for`-loop. It would make more sense for iteration-related functions to appear there, i.e. it would be more logical for the top suggestions to be `begin` and `end`. Of course, we can come up with a couple of hand-written rules to describe this particular situation. However, we need to use a statistical approach to capture such a dependency in the most general form (for any function, type, and context).



**Fig. 1.** An example of a context-sensitive completion. The context of a code affects the list of candidates raising only the most probable ones.

#### 3.1 AST Nodes

In this paper, we use language models to learn and generalize such rules from a user’s code. As mentioned before, several studies have shown that language models can be used effectively for programming languages. Most of these works use tokens as words in language models. Tokens are the result of lexical analysis, the lowest level of program analysis, and consequently, represent shallow information about the code. On the other hand, we want to go one step further, and use AST, which represents a deeper approach. It provides richer syntactic and semantic information.



**Fig. 2.** An example of a depth-first traversal of AST that provides a sequential order.

The n-gram model works on top of sequential structures, but the structure of AST is tree-like. To “straighten” AST, we traverse it in a way similar to the order of corresponding tokens. We select depth-first order with a rule of visiting leftmost children first. Figure 2 shows an example of our traversal. We use AST node types as words in an n-gram model, which is always of a fixed length for any particular programming language. For example, the  $i$ -th word’s context  $\gamma_n(i)$  might have the following form: `FuncDecl BinaryExpr VarDecl UnaryExpr`.

Our goal, though, is to predict the correct function to call. To do so, we need to collect statistics about actual calls. For each n-gram that finishes with a call expression, we store every function’s number of occurrences. To illustrate, the example from Fig. 1 might have the following n-gram: `ForStmt AssignmentExpr VarDecl VarDecl begin`. As a result, the size of dictionary  $\mathcal{D}$  is defined by the number of all types of AST nodes for the given language and the number of all functions in the user’s project.

### 3.2 Candidate Sorting

If  $\gamma(m) = w_{m-n+1}, \dots, w_{m-1}$  is a context at point  $m$ , and  $C(m)$  is a set of all functions that are semantically possible to be called at point  $m$ , then the language model defines a mapping  $P(\cdot|\gamma(m)) : C \mapsto \mathbb{R}_{[0,1]}$ , and a binary relation  $\succ$  on the set  $C(m)$ :

$$\forall a, b \in C(m) \rightarrow a \succ b \Leftrightarrow P(a|\gamma(m)) > P(b|\gamma(m)) \quad (10)$$

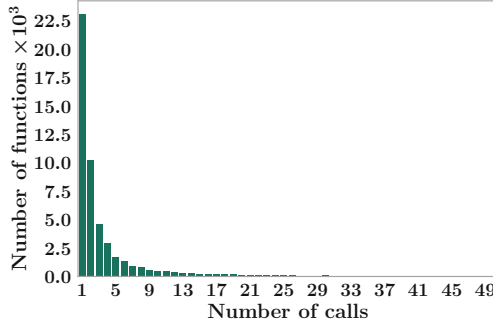
Relation  $\succ$  defines a linear order on  $C(m)$ . We denote  $\{\hat{c}_i\}$  as the ordered set of completion candidates, where  $\forall i, j \in \mathbb{N}_{[1,|C(m)|]} : i > j \Leftrightarrow \hat{c}_i \succ \hat{c}_j$ . We refer to the set  $\{\hat{c}_i\}$  as a completion list, and say that  $\hat{c}_i$  is the  $i$ -th completion candidate.

## 4 Smoothing

According to the *law of large numbers*, we need to encounter each possible n-gram a significant number of times in order to give a good estimate for corresponding probabilities. This represents a perfect world scenario. However, according to  $|\mathcal{D}^n| = |\mathcal{D}|^n$ , the number of possible n-grams grows exponentially with the growth of  $n$ . Because of that, in the real world, some n-grams appear rarely or do not appear in a sample at all. The estimated probability for those n-grams would be  $\approx 0$ , which is not a good estimate. This problem is usually referred to as a *data sparsity problem* [4]. The target of *smoothing* is to construct a better estimate for n-grams that we did not encounter without having to use any additional data [4, 6, 16].

### 4.1 Interpolated Model

A very typical distribution of function calls is shown in Fig. 3. Almost 85% of all functions have less than 20 calls,  $\sim 80\%$  have less than 10, and  $\sim 40\%$  have only



**Fig. 3.** The distribution (The actual distribution’s tail is much longer, reaching to the thousands.) of functions over number of calls on the LLVM project.

one call. This is a restriction that cannot be overcome because it is not possible to gather more data. To ease the effects of the restriction, we use *Jelinek-Mercer* smoothing [12]. According to Chen and Goodman [4], it is better for small test samples. *Jelinek-Mercer* smoothing makes use of lower order models including them in the target estimate. Because of this, the resulting model is often referred to as an *interpolated model*. The target estimate for an interpolated n-gram model is defined by the following formula:

$$\hat{P}_{\text{JM},n}(w_i, \boldsymbol{\lambda}) = \sum_{j=1}^n \lambda_j \hat{P}_j(w_i) \quad (11)$$

where  $\hat{P}_j(w_i)$  is an estimate of the probability for  $w_i$  to appear in the current context according to a j-gram model, and  $\forall j \in [1, n] \rightarrow \lambda_j \in [0, 1] : \sum_{j=1}^n \lambda_j = 1$ .

Many widely-used smoothing techniques are interpolated models and differ in defining the  $\boldsymbol{\lambda}$  parameter [11, 14, 24]. In our work, we pick a constant vector for  $\boldsymbol{\lambda}$ . For evaluation, we use the following three choices:

1. Equal coefficients  $\boldsymbol{\lambda}^=$ :  $\forall j \in [1, n] \rightarrow \lambda_j = \frac{1}{n}$
2. Exponential coefficients  $\boldsymbol{\lambda}^{\text{exp}}$ :  $\forall j \in [2, n] \rightarrow \lambda_j = \frac{1}{2^{n-j+1}}$  and  $\lambda_1 = \frac{1}{2^{n+1}}$
3. Optimal coefficients  $\boldsymbol{\lambda}^*$

## 4.2 Optimal Coefficients

We define optimal coefficients as a minimizing parameter for the following function:

$$L(\boldsymbol{\lambda}, k) = \sum_{c \in \mathcal{C}} \xi(c, \boldsymbol{\lambda}, k) \quad (12)$$

$$\xi(c, \boldsymbol{\lambda}, k) = \begin{cases} \Delta(c, \boldsymbol{\lambda})^2, & \text{if } \Delta(c, \boldsymbol{\lambda}) < k \\ k^2, & \text{otherwise} \end{cases} \quad (13)$$



where  $\Delta(c, \lambda) = i$ , and  $i \in \mathbb{N} : c = \hat{c}_i(\lambda)$ . For a set of calls  $\mathbb{C}$ , function  $L$  defines a cumulative penalty for putting an actual call  $c$  in a different position than first. Parameter  $k$  is a visibility area, which means that positions with an index bigger than  $k$  are penalized with the same value. The following example demonstrates why this is important.

Let the set  $\mathbb{C}$  contain only two calls  $c_1$  and  $c_2$ , where each call has a 100 candidates for completion. Let us also consider that we have some initial vector  $\lambda_1$ . For this vector, our model puts  $c_1$  in the 80th place, and  $c_2$  in the 10th place. Imagine that we have two options of improving the initial vector:  $\lambda_2$  and  $\lambda_3$ . For  $\lambda_2$  our model improves the position of  $c_1$  from 80th to 60th place, for  $\lambda_3$  it improves the position of  $c_2$  from 10th to 4th. The question is: which improvement is better? Without a visibility area, it would be  $\lambda_2$  because it gives better improvement in terms of positions, but this change would not even be noticed by the user. Vector  $\lambda_3$ , on the other hand, makes  $c_2$  visible for the user (if we assume that she can see the top 5 functions in the completion list). In a model with a visibility area, this improvement is better. In summary, the  $k$  parameter allows only reasonable improvements.

To minimize the penalty function, we use a *stochastic gradient descent*. Function  $L$  is not differentiable because we don't have an analytic form for  $\Delta(c, \lambda)$ . Therefore, we are limited to use an empirical estimation of  $L$ 's gradient.

### 4.3 Kneser-Ney Smoothing

Kneser-Ney smoothing [15] is considered to be the most efficient smoothing technique [4]. Its main idea is usually explained with the ‘‘San Francisco’’ example. Let us imagine that ‘‘San Francisco’’ appears in the text many times. Because of this, the separate words ‘‘San’’ and ‘‘Francisco’’ are unigrams with high probabilities. Bigram ‘‘reading Francisco’’, on the other hand, has never been encountered in the text. Using a unigram model to smooth its probability, we would incorrectly conclude that ‘‘reading Francisco’’ should have a high probability because ‘‘Francisco’’ is a common word. This case demonstrates why it's better to estimate unigram probability by the number of different words it follows, instead of the number of occurrences in the text. Since the word ‘‘Francisco’’ appears only after ‘‘San’’, ‘‘reading Francisco’’ should receive a low probability. The idea can be generalized for an n-gram model and described by the following formula:

$$\hat{P}_{KN}^n(w|\gamma_n(i)) = \frac{\max(\sum_{s \in \mathcal{S}_n} \mathbf{1}_{s=\gamma_n(i),w} - \delta, 0)}{\sum_{w' \in \mathcal{D}} \sum_{s \in \mathcal{S}_n} \mathbf{1}_{s=\gamma_n(i),w'}} + \delta \frac{|\{w' : \gamma_n(i)w' \in \mathcal{S}_n\}|}{\sum_{w' \in \mathcal{D}} \sum_{s \in \mathcal{S}_n} \mathbf{1}_{s=\gamma_n(i),w'}} \hat{P}_{KN}^{n-1}(w|\gamma_{n-1}(i)) \tag{14}$$

$$\hat{P}_{KN}^1(w) = \frac{|\{w' : w'w \in \mathcal{S}_2\}|}{|\{(w', w'') : w'w'' \in \mathcal{S}_2\}|} \tag{15}$$

## 5 Implementation

An experimental system was implemented for C/C++ language using Clang libTooling API<sup>1</sup> for code parsing purposes. To capture the correct compiler options and get the whole list of source files, we use CMake and CMake's *compilation database*<sup>2</sup>. For each file, we build AST and collect n-gram frequencies.

For storage, we use suffix trees [23], which provide fast access to collected statistics ( $\mathcal{O}(n)$  for n-gram model). The suffix tree is the best candidate for our needs because it provides access to all lower order models [13]. Thus they grant a compact and memory-efficient way to store data for interpolated models.

To optimize the penalty function  $L$  and retrieve  $\lambda^*$ , we use the GNU Scientific Library [7].

## 6 Evaluation

Empirical evaluation was held on the following open source projects: LLVM+Clang<sup>3</sup>, MySQL<sup>4</sup>, OpenCV<sup>5</sup>, and Caffe<sup>6</sup>.

### 6.1 Theta Function

To measure the precision of our models, we need to use a function that reflects how good our completion system works. It should be based not only on the fact that the target function is ranked as a top candidate, but also on achieving a position visible to the user. The previously defined penalty function  $L$  could be what we are looking, but its values are good only for comparing models, and not for getting an impression of how good the overall prediction is. For this purpose, let us consider the following function:

$$\Theta(m) \doteq \frac{1}{|\mathbb{C}|} \sum_{c \in \mathbb{C}} \sum_{i=1}^m \delta_{c\hat{c}_i} \quad (16)$$

where  $\mathbb{C}$  is a set of all calls,  $c$  is a particular function call,  $\hat{c}_i$  is a  $i$ -th candidate according to an order defined by  $\succ$  binary relation, and  $\delta$  is the Kronecker delta.

$$\sum_{i=1}^m \delta_{c\hat{c}_i} = \begin{cases} 1, & \text{if } c \text{ is among first } m \text{ candidates} \\ 0, & \text{otherwise} \end{cases} \quad (17)$$

<sup>1</sup> <https://clang.llvm.org/docs/LibTooling.html>.

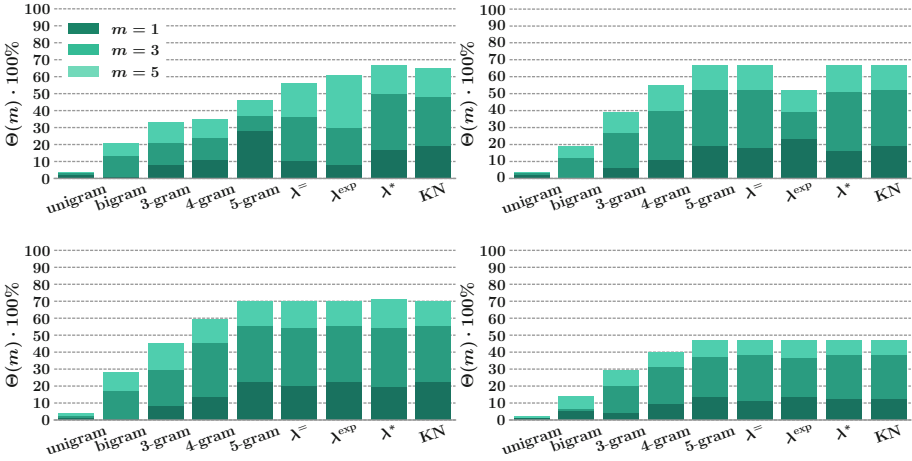
<sup>2</sup> <https://clang.llvm.org/docs/JSONCompilationDatabase.html>.

<sup>3</sup> <http://www.llvm.org/>.

<sup>4</sup> <https://www.mysql.com/>.

<sup>5</sup> <http://www.opencv.org/>.

<sup>6</sup> <http://www.caffe.berkeleyvision.org/>.



**Fig. 4.** Evaluation results on open-source projects: LLVM (top-left), MySQL (top-right), OpenCV (bottom-left), and Caffe (bottom-right)

As a result,  $\Theta$  defines a natural ratio of “guessed” calls to the total number of calls.  $\Theta$  depends explicitly on  $m$  and implicitly on a set  $\mathbb{C}$ . As far as we use  $\Theta$  as a measure of effectiveness for different models, it’s better to use data that was not a part of a sample. Because of this fact, each project was randomly split into three parts: 80% for model construction, 10% for  $\lambda^*$  calculation, 10% for testing.

### 6.2 Results

Figure 4 shows the evaluation results. Alphabetical ordering gives less than 1% for each project and each  $m$ , and thus is not included. The unigram model can be interpreted as a sorting of candidates by popularity, which is a similar approach to [10]. Average time spent on completion list construction is  $2.5 \times 10^{-6}$  sec, which implies that our system can be used as a real-time completion system.

Figure 4 also shows that the best results are bounded from above. The upper bound is different for every project and can be related to the limitation shown by Fig. 3. Even with smoothed models, it’s impossible to estimate the probability of a function that had not been called before. If a function has only one call, then the moment when we try to predict it, we have no data to do so. This limitation cannot be overcome without more deeply integrating semantic information into the model. On the bright size, our smoothed models have managed to reach maximum performance in terms of this limitation:

$$\Theta(5) \approx 1 - \frac{|\mathbb{F}_1|}{|\mathbb{C}|} \tag{18}$$

where  $\mathbb{F}_1$  is a set of functions that are called only once.

## 7 Conclusion and Further Work

In this paper, we presented a new approach to improve the precision of code completion. It is based on statistical language models and uses a user's project as the input data to build the model. We use AST nodes as words for the model, which gives us rich syntactic and semantic information about the code. Due to the fact that AST is constructed for all programming languages, our approach can be applied to any language. Our evaluation shows that the method achieves better results than similar methods. Average time spent on completion for each call is much less than 1 s, and this makes our system applicable for every day use.

There are several directions that further work could take. Our approach "straightened" AST to get a sequential order, but AST is a tree with a significant amount of information stored in its structure. The process of straightening inevitably leads to a loss of information, so future methods should make an attempt to avoid this limitation. One possible way is to use syntactic dependency based n-grams [21].

Another direction would be to add more details in n-grams. We used AST nodes as clusters and used names only for function calls, but AST also contains information about types, variables and operations. Future work should develop a method for integrating this data into the model. A study by Gao et al. [5] is a good place to start, as it shows how to unify a clustering approach with a more detailed one.

However, the most promising direction is to use a language-specific analysis to give better probability estimates. As mentioned earlier, this could solve the problem caused by functions with only one call, which was a major limitation for our system. Raychev et al. [19] presented a synthesis of program analysis and statistical modeling. They use Steensgaard alias analysis [22] to improve results. While this method was not appropriate for our purposes, it is still a good example of how these two worlds can be merged to achieve better results.

## References

1. Stack Overflow developer survey 2016 results. <http://stackoverflow.com/research/developer-survey-2016>
2. Amann, S., Proksch, S., Nadi, S., Mezini, M.: A study of visual studio usage in practice. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 124–134. IEEE (2016)
3. Bruch, M., Monperrus, M., Mezini, M.: Learning from examples to improve code completion systems. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2009), pp. 213–222. ACM, New York (2009). <http://doi.acm.org/10.1145/1595696.1595728>
4. Chen, S.F., Goodman, J.: An empirical study of smoothing techniques for language modeling. In: Proceedings of the 34th Annual Meeting on Association for Computational Linguistics (ACL 1996), pp. 310–318. Association for Computational Linguistics, Stroudsburg (1996). <http://dx.doi.org/10.3115/981863.981904>

5. Gao, J., Goodman, J., Miao, J., et al.: The use of clustering techniques for language modeling-application to Asian languages. *Comput. Linguist. Chin. Language Process.* **6**(1), 27–60 (2001)
6. Goodman, J.T.: A bit of progress in language modeling. Technical report (2001)
7. Gough, B.: *GNU Scientific Library Reference Manual*, 3rd edn. Network Theory Ltd., Bristol (2009)
8. Han, S., Wallace, D.R., Miller, R.C.: Code completion from abbreviated input. In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pp. 332–343. IEEE Computer Society, Washington (2009). <http://dx.doi.org/10.1109/ASE.2009.64>
9. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pp. 837–847. IEEE Press, Piscataway (2012). <http://dl.acm.org/citation.cfm?id=2337223.2337322>
10. Hou, D., Pletcher, D.M.: An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion. In: *ICSM*, pp. 233–242. IEEE Computer Society (2011). <http://dblp.uni-trier.de/db/conf/icsm/icsm2011.html#HouP11>
11. Hsu, B.J.: Generalized linear interpolation of language models. In: *IEEE Workshop on Automatic Speech Recognition & Understanding (ASRU 2007)*, pp. 136–140. IEEE (2007)
12. Jelinek, F., Mercer, R.L.: Interpolated estimation of Markov source parameters from sparse data. In: Gelsema, E.S., Kanal, L.N. (eds.) *Proceedings Workshop on Pattern Recognition in Practice*, pp. 381–397. North Holland, Amsterdam (1980)
13. Kennington, C.R., Kay, M., Friedrich, A.: Suffix trees as language models. In: Calzolari, N., Choukri, K., Declerck, T., Doan, M.U., Maegaard, B., Mariani, J., Moreno, A., Odijk, J., Piperidis, S. (eds.) *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC 2012)*. European Language Resources Association (ELRA), Istanbul, May 2012
14. Klakow, D.: Log-linear interpolation of language models. In: *Proceedings of ICSLP 1998*, pp. 1695–1698 (1998)
15. Kneser, R., Ney, H.: Improved backing-off for m-gram language modeling. In: *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Detroit, Michigan, vol. 1, pp. 181–184, May 1995
16. Manning, C.D., Schütze, H.: *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge (1999)
17. Murphy, G.C., Kersten, M., Findlater, L.: How are java software developers using the eclipse IDE? *IEEE Softw.* **23**(4), 76–83 (2006). <http://dx.doi.org/10.1109/MS.2006.105>
18. Omar, C., Yoon, Y., LaToza, T.D., Myers, B.A.: Active code completion. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pp. 859–869. IEEE Press, Piscataway (2012). <http://dl.acm.org/citation.cfm?id=2337223.2337324>
19. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. *SIGPLAN Not.* **49**(6), 419–428 (2014). <http://doi.acm.org/10.1145/2666356.2594321>
20. Robbes, R., Lanza, M.: How program history can improve code completion. In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pp. 317–326. IEEE Computer Society, Washington (2008). <http://dx.doi.org/10.1109/ASE.2008.42>

21. Sidorov, G.: Syntactic dependency based n-grams in rule based automatic English as second language grammar correction. *Int. J. Comput. Linguist. Appl.* **4**, 169–188 (2013)
22. Steensgaard, B.: Points-to analysis in almost linear time. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1996)*, pp. 32–41. ACM, New York (1996). <http://doi.acm.org/10.1145/237721.237727>
23. Weiner, P.: Linear pattern matching algorithms. In: *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973) (SWAT 1973)*, pp. 1–11. IEEE Computer Society, Washington (1973). <http://dx.doi.org/10.1109/SWAT.1973.13>
24. Witten, I.H., Bell, T.C.: The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. *IEEE Trans. Inf. Theor.* **37**(4), 1085–1094 (2006). <http://dx.doi.org/10.1109/18.87000>

# Using the Subject Area Ontology for Automating Learning Processes and Scientific Investigation

Dmitry Shachnev<sup>1</sup>(✉) and Dmitry Karpenko<sup>2</sup>

<sup>1</sup> Faculty of Mechanics and Mathematics,  
Lomonosov Moscow State University, Moscow, Russia  
mitya57@mitya57.me

<sup>2</sup> Institute of Complex Systems Mathematical Research,  
Lomonosov Moscow State University, Moscow, Russia  
dskarpenko@gmail.com

**Abstract.** This paper presents the ways and the methods to build and populate the ontology, based on the taxonomy of the subject area, which can then be used to automate the learning processes. The applications include: analysis of educational and teaching activity of students and lecturers, contextual analysis and thematic search in big data systems, analysis of texts including the anti-plagiarism protection, modeling the learning processes and generating various content such as teaching plans and tests. The ontology is also used to form the unified semantic network of the research center.

The ontology is populated by scientists and users working on different branches of science, using the ontology editor developed by the authors. The editor is built as part of ISTINA (Intellectual System for Thematic Investigation of Scientometrical Data), the current research information system used in Lomonosov Moscow State University. The core concepts and relations in each branch of science are verified by experts on that branch.

**Keywords:** Scientometrics · Ontology · Subject area · Taxonomy  
Learning process · Tests generation · Knowledge map

## 1 Goals of the Work

The goal of this work is to develop a way to improve the scientific investigation and learning processes in a high school, by formalizing the subject area and allowing for effective actualization of it. To achieve this, we develop the algorithms and the technical means that would allow us to build such formalization in a form of an ontology, control the quality of it and use it for generating the students' individual learning trajectories and work plans.

---

D. Karpenko—The authors would also like to thank Sergey Artamkin and Gennady Bogopolsky for their help with preparing this paper.

## 1.1 Previous Works in This Area

The problem of formalizing the learning process is quite popular in Russia, because the education system here is more centralized and less oriented on the concrete lecturers. Some research on using an ontology for this task was performed in Lomonosov Moscow State University [3, 7], in Pirogov Russian National Research Medical University (RNRMU) [2, 6] and in the A. P. Ershov Institute of Informatics Systems of the Siberian Branch of Russian Academy of Sciences [8].

## 2 The ISTINA System

ISTINA (the Russian abbreviation for “Intellectual System for Thematic Investigation of Scientometrical Data”) [4, 5] is the current research information system (CRIS) which is used in Lomonosov Moscow State University and in several institutes of the Russian Academy of Sciences. It stores various results of scientific and educational activity of employees, such as articles, conference talks, patents, research projects, lecture courses, students’ guidance, etc. It is integrated with international systems such as Web of Science and Scopus. Figure 1 shows a profile of a user in ISTINA.



**Dmitry Shachnev** user responsible

Moscow State University, Faculty of Mechanics and Mathematics, Department of Mathematics, Chair of Computational Mathematics, PhD student, since October 1st, 2015

Coauthors: Alexander Kozitsyn, Sergey Afonin

2 articles, 2 conference talks, 4 research projects

IstinaResearcherID (IRID): 8920536

Activity  BBTeX style: **normal** | GOCT | plain | abbrev | acm | alpha | amsalpha | amsplain | apalike | ieee | siam

---

**JOURNAL ARTICLES**

2016 Программные механизмы агрегации данных, основанные на онтологическом представлении структуры реляционной базы наукометрических данных  
Афонин С.А., Козицын А.С., Шачнев Д.А.  
in journal *Программная инженерия*, publisher *Новые технологии (М)*, № 9, pp. 408-413 DOI

---

**COLLECTION ARTICLES**

2016 Использование онтологического представления структуры реляционной базы для агрегации наукометрических данных  
Шачнев Д.А., Афонин С.А., Козицын А.С.  
in collection *Научный сервис в сети Интернет. Труды XVIII Всероссийской научной конференции*, publisher *ИПТМ им. М.В. Келдыша РАН (Moscow)*, pp. 58-63 DOI

**Fig. 1.** Profile of a user in ISTINA system.

One of the key features of ISTINA is the ability to build a list of works for an employee using a certain formula, with every work having a weight according to that formula. A formula is a set of lines, where each line defines a set of works and a function to build the weight. For example, a line can say that articles in a journal having a Scopus impact-factor get a weight equal to  $N \times \text{journal IF} \div \sqrt{\text{number of coauthors}}$ . It is possible to add multiple filters and restrictions,



such as “select all oral talks on international conferences where the worker was the presenter”. A work is included into the result if it matches at least one formula line; if it matches several lines the one which yields the maximum weight is used. Internally these formulae are stored as JSON, where each line is a tuple of (category of works, restrictions, parameter to use as initial weight, numeric multiplier, modifier functions to apply). A formula can be evaluated for a single employee, forming a set of ranked works, or for a department, which gives a ranked list of employees where for each employee a sum of their works’ weights is given.

The formulae evaluating system uses its own very simple ontology structure. A category in a formula defines a class of works, and a relation between a worker and a work, for example “course–author”, “course–lecturer”, or “project–responsible implementer”. Each category has a set of properties, either numeric or boolean. For example, for a course being read these are its duration in weeks, number of academic hours per week, and number of students. There is a JSON-like structure which maps the categories and the properties onto the underlying relational database structure. This mapping is used to generate the SQL queries for each line of a formula. The internal structure of the formulae and the algorithms used in the SQL generator are described in detail in [1].

### 3 Structure of the Ontology

The most important part of the ontology is the taxonomy, which is the hierarchy of the subject areas that are used in a high school. The rest of the ontology is built around the taxonomy. The ontology consists of multiple layers, listed below.

- The core ontology, which is in general immutable and can only be edited by the administrators. It consists of the meta-ontology, which contains some basic concepts and relations, and the items which have a special meaning throughout the ISTINA system, most importantly the ontology used in the formulae.
- The teaching plans and the hierarchy of the disciplines. The work teaching plan of the chair describes one or several disciplines; each discipline consists of several modules, and each module has several topics and subtopics.
- The terms from each branch of science and the relations between them.
- Any other concepts that are needed to build the relations between the layers mentioned above.

The ontology concepts can usually be divided into classes and their instances. These work like classes and instances in object-oriented programming. For example, “Stokes’ theorem” is an instance of class “theorem”, and “Linear algebra” is an instance of class “discipline”. Everything except the core ontology is freely editable by all users, but the changes can be moderated by experts in the relevant science areas, which are granted special rights for this.

## 4 Building the Ontology

From organizational point of view the process of building the ontology can be split into two stages. During the first stage, the ontology is tightly bound to the teaching process. Each chair or department develops its own part of the ontology, which is usually bound to the courses that are being read by that chair. There can be no deviations within the chair, to make it easier to merge the resulting ontologies afterwards. On the second stage, all separate parts of the ontology are joined together, and each worker can maintain his own changes to the structure of the chair.

### 4.1 The Ontology Editor

The ontology editor can work in two different modes: the standalone mode and the inline mode. In the standalone mode the editor has a row of tabs, which are used to switch between pages. The first tab is called “Edit course structure and terms” and is shown on Fig. 2. On this tab one can enter the hierarchical structure, and then for every topic and subtopic in this structure add the terms related to it.

The screenshot shows the 'Edit course structure and terms' tab. On the left is a tree view:

- Chemistry of biomolecules and nanosystems
  - Bioorganic chemistry
    - + Properties and biologically important reactions of poly- and heterofunctional compounds
    - + Structure of monosaccharides, glycosides, disaccharides
    - + **Heterocyclic compounds, nucleotides and nucleic acids**
    - + Add a topic
  - Physics and chemistry of organic nanosystems and biopolymers
    - + Colloidal surfactants
    - + Peptides, α-amino acids
    - + Chromatography
    - + Add a topic
    - + Add a module
    - + Add a discipline

On the right, the selected term 'Nucleotides' is shown with a 'term' label and a 'click to edit' button. Below it is a description: 'Nucleotides are the phosphate esters of nucleosides. Free nucleotides, in particular ATP, cAMP, and APP, play an important role in the intracellular energy and information processes, and are parts of many nucleic acids and coenzymes.' A URL is provided: <http://istina.msu.ru/ontology/bioorganic-chemistry/nucleotides>. Below this, the term 'Heterocyclic compounds' is shown with a 'term' label and a 'click to edit' button. Its description is: 'Heterocyclic compounds, or the heterocycles, are cyclic compounds which have at least two different elements in their rings. Usually one of the elements is carbon, although such compounds may be inorganic.' A URL is provided: <http://istina.msu.ru/ontology/bioorganic-chemistry/heterocyclic-compounds>. At the bottom right, there is a '+ Add a term' button.

Fig. 2. The structure and terms page of the ontology editor.

The other tabs are called *ontology pages*, and are used for entering the relation types and the relations themselves. Each page belongs to the user who created it: everyone can view it, but only the page owner can edit it. The page has a set of concepts associated with it, all relations within this set are automatically shown on the page. While it is recommended to enter the terms on the first tab, there is a built-in interface to add the new terms on the work pages too. There is also a way to split the existing relation into two relations by inserting a new term in the middle.

There is a version control system for pages and their content. It is possible to compare different revisions of the same page, or pages of different users. It is also possible to “fork” a page of different user and make changes to it.

The algorithm for adding terms and relations to the system using the ontology editor is as follows.

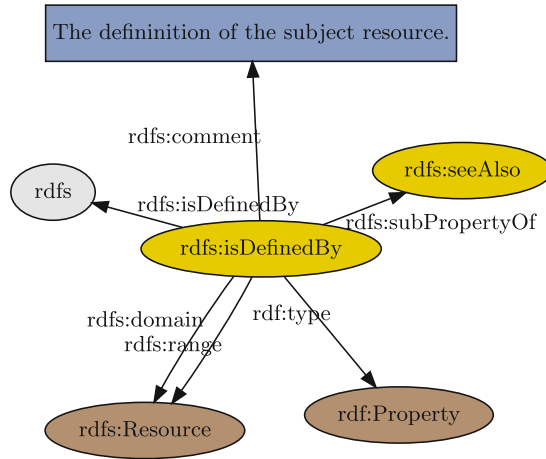
1. On the course structure tab, the relevant topic or subtopic should be selected. If it does not exist, it can be created using the “Add a topic” link. Similar links can be used to create the missing disciplines and modules.
2. New terms should be added using the “Add a term” button. For each term, its name and description should be entered. It is also possible to click on existing terms and edit their names or descriptions.
3. If a term has an equivalent in one of the projects that are part of the Linking Open Data cloud, it is possible to add the URIs for those remote terms, and the links will be created.
4. Either a new page should be created, or an existing one should be opened. The existing pages usually represent the lowest in the hierarchy branches of science, or the instances of a particular class. The user can also save some page and reopen it later.
5. If it was a new page, some existing terms should be loaded to that page first. This can be done using the search field. If there are multiple objects with the same name, the object classes are shown in parentheses for disambiguation. If some of the added terms have any relations between each other, these relations are automatically shown on the page. It is possible to filter the relations types by clicking on checkboxes with these types in the side panel.
6. Relations between terms should be drawn. Dragging the cursor from one item to the other one automatically creates a relation and puts it into focus. After that it is possible to choose the relation type, using the same interface that is used for searching for terms. Figure 3 shows how the editor looks like when the `rdfs:isDefinedBy` concept from the core (RDF) ontology is opened.

## 4.2 Technical Implementation of the Ontology Store

The ISTINA system uses a relational database to store all its data and the Django web framework to process requests to the website. Because of the need to integrate the ontology and the existing objects in the relational database, it has been decided to store the ontology in the same database, and not in a separate RDF storage. We have taken advantage of *RDFLib*, a Python library for working with RDF, and have implemented a custom Store backend for RDFLib which uses the Django object-relational mapping to generate the SQL queries to the relational database.

In the database, the following tables are used:

- **Concept** — maps concept URIs to internal numeric IDs;
- **ConceptProperty** — stores literal values of properties;
- **Triplet** — stores (*subject*, *predicate*, *object*) triplets where *subject* and *object* are generic foreign keys: for example, *object* can be a concept, a literal value, or an object that is part of ISTINA system but not part of the ontology;
- **Revision** — stores meta-information about ontology revisions: revision number, user and creation date;
- **Page**, **PageMembership** — stores information about ontology pages, who owns them, and which concepts belong to which pages.



**Fig. 3.** The `rdfs:isDefinedBy` concept and its properties.

The `Triplet` table also has two fields that store data needed for the version control system: `RevCreated` and `RevDeleted`. When a triplet is created, the first field gets a value equal to the revision ID, the second field stays `NULL`. When a user requests to delete the triplet, it is not actually deleted, but the second field is set to the revision ID instead. This way the version control operations become very easy.

- To browse the current state, one can use the `RevDeleted is NULL` expression.
- To browse a state at revision  $N$ : `RevCreated <= N and (RevDeleted is NULL or RevDeleted > N)`.
- To check changes of revision  $N$ : `RevCreated = N or RevDeleted = N`.

RDFLib provides means to import and export data in different formats: in particular, import foreign ontologies in Turtle format. This can be used for importing standard ontologies (such as Dublin Core), or ontologies created in offline editors, such as Protégé. RDFLib also provides a SPARQL endpoint implementation, which automatically converts SPARQL queries to method calls to the Store backend (which are then converted to SQL queries). The result of a SPARQL query can be serialized in several formats, the most important for us being the JSON format documented by the W3 Consortium. The ontology editor performs some SPARQL queries via AJAX, for example, to get a list of all classes.

The changes from the editor (client side application) to the store (server side application) are submitted in form of “change lists”. Every change in a change list can be of the following types: create or remove a concept, create or remove a triplet, add a concept to or remove from a given page. The change list is represented as a JSON array where every item is a tuple specifying the change type and its parameters (for example, the concept URIs). The patch is

processed as a single database transaction: if applying a change fails, then the whole change list is not saved, and the error message is presented to the user.

### 4.3 Automatic Suggestions

The inline mode of the editor is used on the page where new works are added to the ISTINA system. When an annotation is entered into the system, it is parsed using the modified version of Brainsterm algorithm [3], which extracts the potential terms and thesaural relations between them. These suggestions are presented to the work author, who then checks for their correctness and can add them to the ontology in several clicks.

## 5 Applications

### 5.1 Searching for Works and Authors

Each work in the system has some keywords, which are linked to concepts in the ontology. Given the search query, it is analysed, and the most close nodes in the ontology are determined for it as well. Then, for every work in the system, its weight is set to the distance between two sets of concepts, that is the closest distance between a concept from the keywords and a concept from the search query. In some cases, a logarithmic function of a distance can be used instead of the plain value of distance. Technically this is implemented by maintaining a cache which stores the closest pairs of keywords throughout the system, and closest concepts for the keywords for every work in the system.

Given the keywords, we can also find the authors which have the biggest number of works in an area determined by these keywords. First, based on the works found, the set of candidate authors is determined. Then, for this set, a formula like ones described in Sect. 2 is evaluated, where the weights of works calculated with the formula are multiplied by the weights corresponding to the search. This approach allows us to perform search in accordance to the users' needs: for example, one can search for authors that have most articles on given subject in journals with high impact-factor only.

As the modules and topics have terms attached to it, the system can also use that to find the literature to help students with these topics.

### 5.2 Anti-plagiary Text Analysis

We are currently experimenting with the ways to improve the detection of plagiarism. The classical approach of finding similar blocks of text is more effective when it is combined with comparing the ontologies that were generated from the texts. While the automatic suggestions formed by the parser are not always smart enough to generate a proper ontology, they can be used as an invariant: when structure of a sentence changes slightly, the structure of the ontology usually remains the same. This allows us to find the semantically similar works, which may indicate that either these are works in the same field of science, or one of the works was copied without proper citing.

### 5.3 Generating Collections of Test Exercises

The test exercises are generated automatically based on the ontology that is linked to a discipline or to a module. The system can generate the collections of test exercises, where each collection covers the whole discipline, and the exercises in all collections differ from each other.

There are several kinds of exercises, which correspond to different kinds of relations. For example, a possible task for a symmetric relation is to match pairs of terms, and a task for an asymmetric relation is to order the terms into a sequence based on that relation.

Examples of generated test questions are given below.

1. Which of these are parts of *Hemispherium cerebri*?

- |  |   |
|--|---|
| <input type="checkbox"/> <i>Lobus frontalis</i>          | <input type="checkbox"/> <i>Lobus occipitalis</i>             |
| <input type="checkbox"/> <i>Insula (lobus insularis)</i> | <input type="checkbox"/> <i>Substantia perforata anterior</i> |

2. Which of these are on the surface of *Facies superolateralis hemispherii cerebri*?

- |   |   |
|---|---|
| <input type="checkbox"/> <i>Gyrus postcentralis</i> | <input type="checkbox"/> <i>Gyri orbitales</i>                    |
| <input type="checkbox"/> <i>Gyrus precentralis</i>  | <input type="checkbox"/> <i>Gyrus occipitotemporalis medialis</i> |
| <input type="checkbox"/> <i>Gyrus rectus</i>        |   |

3. Put the following anatomic formations in the order from the innermost to the outermost (put numbers 1–4 into the fields):

- |  |                           |
|--|---------------------------|
| __ <i>Caput nuclei caudati</i>           | __ <i>Putamen</i>         |
| __ <i>Crus anterius capsulae interna</i> | __ <i>Capsula externa</i> |

The test exercises generated from the ontologies have the following benefits: (a) they are formed automatically; (b) it is impossible for students to learn the correct answers, because the questions are generated randomly and there are up to several thousands possible variants; (c) the students should learn the relations between terms, not only the definitions of terms themselves; (d) it is possible to analyze the test results of a student or of a group, and form a new test based on topics where the result was the worst (see the next subsection for details).

Another possible task is asking the students to draw the parts of ontologies on some topic themselves. They can use the ontology editor for this task, however they will work in an isolated environment, and the concepts from the main ontology would not be available to them. Then the two ontologies can be compared, and wrongly entered concepts and relations can be detected.

### 5.4 Building the Knowledge Map for a Student

Based on the test results, we can build knowledge maps for individual students or for groups of students. Each term in the test ontology can have one of four states for every student: (a) learned correctly; (b) learned incorrectly; (c) not learned; (d) not checked in the test. Each link between terms can have one of three states: (a) link inserted correctly; (b) link inserted incorrectly; (c) link not inserted by the student, although it was part of the original ontology.

The knowledge map is generated from the original ontology by applying colors corresponding to the states of terms and links. An example is shown in Fig. 4, based on the data kindly granted by Dmitry Karpenko, Gennady Bogopolsky and Alexander Sokolov. Figure 5 shows how the ontological approach to the student progress map is better than the classical ways to form such a map, and provides an example of how the disciplines can intersect and form the knowledge map together.

### 5.5 Generating the Individual Learning Trajectories

The individual learning trajectories for students are generated based on the knowledge maps. The accent on study is given to the topics and subtopics where the percent of correctly learned terms and links in the knowledge map is lower. As the map is updated, the learning trajectories can be shifted to different subtopics.

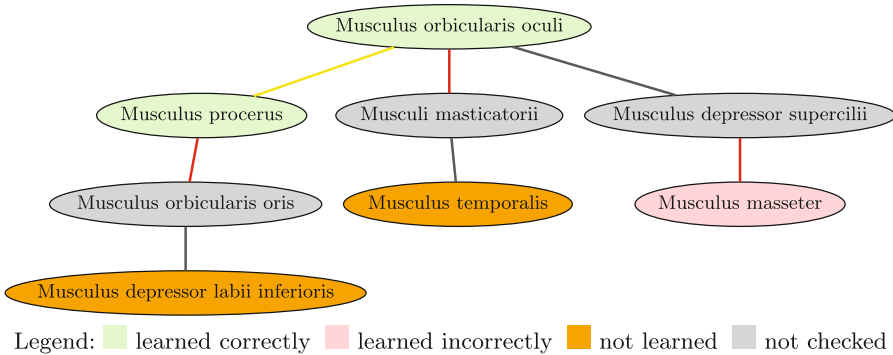
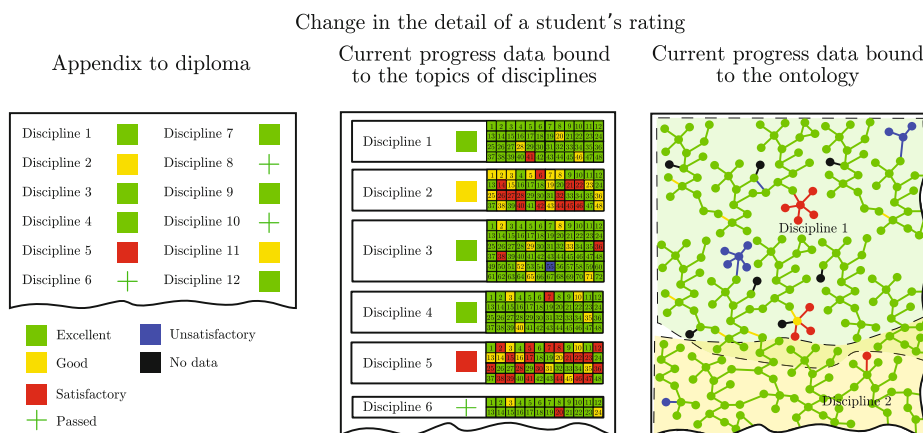


Fig. 4. A fragment of knowledge map based on a test about myology.

### 5.6 Adding More Detail into the Working Programs of High Schools

The Russian standard for teaching materials defines the documents that every high school needs to provide for each teaching program. The key part of every document is the glossary, which is generated automatically from the ontology. The key benefit of using the ontology is the ability to re-use the same work for different courses, with the fixes and updates automatically propagating to each document without any need of manual copying.



**Fig. 5.** Comparison of classical and ontological approaches to the student progress map.

## 6 Conclusion

The developed technology allows the high schools to solve a number of practical problems occurring there. The effectiveness is demonstrated with an example of generating collections of test exercises. For each discipline, an “onto-discipline” is created, the three benefits of which are:

- formalization of the subject area;
- ways for effective actualization of the subject area;
- integration with other onto-disciplines.

## References

1. Afonin, S., Kozitsyn, A., Shachnev, D.: Software mechanisms for scientometrical data aggregation based on ontological representation of the relational database structure. *Softw. Eng.* **7**(9), 408–413 (2016). <http://novtex.ru/prin/eng/10.17587/prin.7.408-413.html>
2. Bogopolsky, G., Karpenko, D., Rauzina, S., Zarubina, T., Tikhonova, T.: Knowledge management within the medical university. *Stud. Health Technol. Inf.* **213**, 107–110 (2015)
3. Golomazov, D.: Methods and tools for managing scientific information with ontologies. Ph.D. thesis, Lomonosov Moscow State University, February 2012. <https://istina.msu.ru/dissertations/1857980/>
4. Lomonosov Moscow State University: The ISTINA website. <https://istina.msu.ru/>
5. Sadovnichiy, V., Afonin, S., Bakhtin, A., Bukhonov, V., Vasenin, V., Gankin, G., Gasparyants, A., Golomazov, D., Itkes, A., Kozitsyn, A., Tumaykin, I., Shapchenko, K.: The Intellectual System of Thematic Investigation of Scientometrical Information (“ISTINA”). Lomonosov Moscow State University (2014). <https://istina.msu.ru/publications/book/7375366/>



6. Tikhonova, T., Sutyagin, P., Rauzina, S.: Ontologies in learning of human anatomy in institute of higher medical education. *Int. J. Exp. Educ.* **5**(3), 277–280 (2016). <https://www.expeducation.ru/ru/article/view?id=10012>
7. Vasenin, V., Afonin, S., Golomazov, D., Kozitsyn, A.: The intellectual system of thematic investigation of scientometrical information (ISTINA). *Inf. Soc.* **1–2**(3), 21–36 (2013). <http://emag.iis.ru/arc/infosoc/emag.nsf/BPA/f04aeb516e101d3d44257be8003c9517>
8. Zagorulko, Y., Zagorulko, G.: Ontology-based technology for development of intelligent scientific internet resources. In: Fujita, H., Guizzi, G. (eds.) *SoMeT 2015*. CCIS, vol. 532, pp. 227–241. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-22689-7\\_17](https://doi.org/10.1007/978-3-319-22689-7_17)

# Runtime Specialization of PostgreSQL Query Executor

Eugene Sharygin<sup>1,2</sup>(✉), Ruben Buchatskiy<sup>1</sup>, Roman Zhuykov<sup>1</sup>,  
and Arseny Sher<sup>1,2</sup>

<sup>1</sup> Institute for System Programming of the Russian Academy of Sciences,  
Moscow, Russia

{eush,ruben,zhroma}@ispras.ru, sher-ars@yandex.ru

<sup>2</sup> Lomonosov Moscow State University, Moscow, Russia

**Abstract.** For computationally intensive workloads, achieving high database performance is in direct correspondence to utilizing CPU efficiently. At the same time, interpretation overhead inherent to traditional interpretive SQL engines gets in the way of optimal CPU utilization.

One solution to this problem is dynamic query compilation, which consists in generating efficient machine code at run time given a particular input query.

Creating a complete query compiler from scratch for an existing database system takes a large amount of development and maintenance effort. Similar results, however, can be obtained more easily using program specialization of a generic query engine with respect to a particular query.

This paper presents intermediate results of applying this approach to the query engine at the core of PostgreSQL database system.

**Keywords:** Dynamic optimization · JIT compilation  
Partial evaluation · Runtime specialization · Query execution  
PostgreSQL · LLVM

## 1 Introduction

One of the central parts of any database system is its query engine, which takes a query from an input channel (usually a network socket) and executes it on the database. This is naturally an interpretive process, and, in fact, in most database systems query engines are implemented as straight query interpreters.

However, interpretation overhead often takes its toll on the overall performance of a database system, which warrants a search for a more efficient query engine implementation.

One usual alternative to interpretation is just-in-time compilation, which has been widely deployed across different domains, including database systems

---

This work is supported by RFBR grant 17-07-00759 A.

[CPS+81, Gre99, KVC10, Neu11, NL14, MB17]. Creating a just-in-time compiler takes large amounts of development effort.

Some developing compiler toolchains strive for minimizing the programmer effort needed in order to arrive at an efficient language implementation, for example, by deriving efficient JIT compilers from specially crafted interpreters [WWS+12] or providing tools to develop compilers in the style of interpreters [RO10], all with little to no guidance from language implementors. However, these techniques are not applicable to existing language interpreters.

Micro-specialization [ZDS12, ZSD12a, ZSD12b] is another approach to optimizing DBMS performance, which consists in replacing particular utility functions with implementations specialized at run time based on hand-crafted templates. However, the applicability and efficiency of this approach is limited due to the lack of automaticity.

For the task of developing a query compiler in an extension to an existing database system, ideally, we would like to keep the source code of its existing interpretive query engine as a reference implementation of the database business logic instead of creating a parallel but different implementation. Given the source code of an interpreter and a new query, the tool would produce efficient machine code that is semantically equivalent to the interpreter, but has superior performance. This would allow to combine interpretive and compiled execution within a single database system and continue to use the interpreter for queries that are not worth the compilation effort. Perhaps more importantly, this would redeem us from the burden of continuing support and maintainance of our newly developed just-in-time compiler as to bringing new features and bug fixes from the existing query interpreter.

Automatic program specialization is exactly that tool. Generally, it is a type of optimizing program transformation that, given a program and some of its input data, generates a residual program that, when given remaining input data, yields the same result as running the original program on all of its input data.

This paper describes the ongoing work in developing a specializer for PostgreSQL [Pos] query executor using LLVM compiler infrastructure [LLVM]. It is organized as follows. Sections 2 and 3 describe online and offline partial evaluation, which are two different specialization methods that we used. Section 4 describes application of runtime specialization to PostgreSQL query engine. Finally, Sect. 5 concludes the paper.

## 2 Online Partial Evaluation for LLVM

Online partial evaluation is the kind of specialization that is based on aggressive constant propagation and loop unrolling.

Online partial evaluator takes a root function and a list of values of the arguments and starts repeatedly evaluating the function's basic blocks until reaching a fixed point. Partial evaluator attempts to evaluate each instruction statically given known values of its operands. If it succeeds in doing so, the instruction is subsequently eliminated and its value is replaced with the constant it has been

evaluated to. This basic procedure can be implemented rather efficiently using the properties of the SSA form which is provided by LLVM.

In contrast to compile-time online partial evaluators such as LLPE [Smo14], a runtime partial evaluator can take advantage of the constant values residing in the process memory. This is implemented by the following transfer function:

$$f_{x=\text{load}(a)}(m) = \begin{cases} m[x \mapsto \text{heap}(m(a))], & m(a) \in \text{Const}_0 \\ m[x \mapsto \perp], & m(a) = \perp \\ m[x \mapsto \top], & m(a) \notin \text{Const}_0 \cup \{\perp\} \end{cases},$$

where  $m: \text{Var} \rightarrow \text{Const} \cup \{\top, \perp\}$  is a lattice value,  $\text{Const}_0 \subseteq \text{Const}$  is a set of addresses of constants in the heap, and  $\text{heap}(c)$  is the value of the memory cell at the given location.

Despite the conceptual simplicity, we faced several problems with online partial evaluation which mostly boil down to the following:

- The results are hard to test, debug, and visualize. Online partial evaluation fuses annotation and specialization into a single run-time phase, which (1) requires a developer to find test queries exercising very specific parts of the code in order to ensure the expected binding-time division and (2) provides no intermediate representation better suited for analysis and visualization than specializer-generated code with its lack of static parts, duplicated dynamic code and unrolled loop iterations.
- Having no means to conduct the analyses ahead-of-time means that continuing development of online partial evaluator inevitably leads to trading complexity and accuracy of partial evaluation for reduced run-time overhead (or vice versa), which is highly suboptimal because the binding-time division does not have to be done at run time since it doesn't require anything but the source code of the interpreter.

### 3 Offline Partial Evaluation for LLVM

In contrast to online partial evaluation, offline partial evaluation proceeds in two distinct phases:

1. At compile time, binding-time analysis (BTA; see Sect. 3.2) takes the root function (the entry point of the interpreter) and the initial argument division, which is a list of binding times (“static” or “dynamic”) per each of its arguments. The values of static arguments are known at specialization time (and not during BTA), while the values of dynamic arguments are only known at query execution time (and not at specialization time). BTA then annotates each instruction as either static or dynamic, which indicates whether the instruction can be completely evaluated at specialization time or not.
2. The annotated program is then specialized at run time (Sect. 3.3) with respect to concrete values of static arguments, which results in a program specialized to those values. Specialization combines execution of static program fragments with generating residual code for dynamic program fragments.

One of the main benefits of offline partial evaluation is the ability to separate these two phases in time: BTA can be conducted ahead-of-time (which makes it a lot easier to test and debug the BTA, and visualize the results), while specialization (directed by annotations in a rather straightforward manner) runs just-in-time.

To our knowledge, this is the first work to describe in detail and implement such a scheme for LLVM IR, although the idea has been proposed before [LC14].

### 3.1 Representation of Binding Times in LLVM IR

As for the intermediate representation BTA and specialization are performed at, we find LLVM IR rather satisfying. It allows a developer to use the wide range of tools, either included in LLVM or external, and to integrate program specialization gradually into an otherwise LLVM IR-centric system (such as our traditional JIT-compiler for PostgreSQL we are basing this work upon [MB17]).

```
define i32 @power.ds(i32 %x, i32 %n) !arg_spec !1 {
entry:
  %n.mod2 = srem i32 %n, 2, !static !2
  %n.odd = icmp eq i32 %n.mod2, 1, !static !2
  br i1 %n.odd, label %odd, label %even

odd:
  %n.1 = phi i32 [ %n, %entry ], [ %n.half, %even.step ], !static !2
  %x.1 = phi i32 [ %x, %entry ], [ %x.sqr, %even.step ]
  %n.dec = add nsw i32 %n.1, -1, !static !2
  %result.prev = tail call i32 @power.ds(i32 %x.1, i32 %n.dec)
  %result.odd = mul nsw i32 %result.prev, %x.1
  ret i32 %result.odd

even:
  %n.2 = phi i32 [ %n.half, %even.step ], [ %n, %entry ], !static !2
  %x.2 = phi i32 [ %x.sqr, %even.step ], [ %x, %entry ]
  %n.positive = icmp sgt i32 %n.2, 0, !static !2
  br i1 %n.positive, label %even.step, label %zero

even.step:
  %x.sqr = mul nsw i32 %x.2, %x.2
  %n.half = lshr i32 %n.2, 1, !static !2
  %n.half.mod2 = and i32 %n.half, 1, !static !2
  %n.half.even = icmp eq i32 %n.half.mod2, 0, !static !2
  br i1 %n.half.even, label %even, label %odd

zero:
  ret i32 1
}
```

Fig. 1. Example of LLVM IR with binding-time annotations

LLVM IR provides a means of expressing domain-specific semantics using the notion of a metadata: each instruction or function can be labeled in an application-specific way, which is exactly what we need to represent binding-time annotations and to link the annotated code of a particular function to its source (so that it can later be discovered by the specializer). Figure 1 shows an example of binding-time annotations as expressed using LLVM IR metadata.

Overall, our BTA operates on the LLVM module containing the source code of the interpreter, and, one by one, annotates each of its functions, in effect creating annotated function variants and linking them to their sources. The result is a fully functional LLVM module, which contains both all the original functions and their annotated variants.

### 3.2 Binding-Time Analysis

BTA described here is polyvariant in functions and monovariant in basic blocks, meaning that it can produce multiple annotated variants for a single function (effectively cloning each function for each group of contexts with matching argument divisions) but it does not clone function's basic blocks or annotate instructions within a single function ambiguously.

For any given function, BTA computes binding-time annotations for each of its instructions (except for control flow transfer instructions, see below). Any instruction's binding time is a function of the binding times of its operands.

The basic rule, called the congruence condition, that drives the analysis is that all instruction users (with the sole exception of the call instruction — see below) of a dynamic value are themselves dynamic by necessity. In order to compute the solution preserving the maximum amount of static information, the algorithm starts from annotating most of the instructions as static and then repeatedly fixes binding times by restoring this property across SSA edges (by changing some binding times back to dynamic), until reaching the fixed point.

**Annotating Memory Access Instructions.** In addition to the binding time of the address operand, annotating a memory access instruction also needs the binding-time type of a corresponding data type, which is a supplementary piece of annotation indicating which fields of a data structure, if any, can be loaded at specialization time. If both the address operand and the accessed field are static, then the load is safe to be annotated static.

```

%ExprState = type {
  static i32, ;tag
  static i8, ;flags
  dynamic i8, ;resnull
  dynamic i64, ;resvalue
  static %TupleTableSlot*, ;resultslot
  static %ExprEvalStep*, ;steps
  static %Expr* ;expr
}

```

**Fig. 2.** Binding-time type annotations

Binding-time types are expressed in a dialect of LLVM IR (not in LLVM IR proper because the latter does not currently allow metadata on types). See example in Fig. 2.

The reason these annotations need to be supplied rather than computed is not only predictability and simplicity of the approach, but mainly that the latter would require a whole-program alias analysis which would lead to overly conservative results since the query interpreter we apply the specializer to performs multiple calls to external database management functions which are out of scope of specialization.

**Annotating Control Transfer Instructions.** Control flow transfer instructions such as branch or return are always annotated dynamic for the reason that doing the opposite would effectively mean combining several blocks into one at specialization time, which is not only of little value because the post-specialization code is optimized by LLVM anyway, but also actively harmful since it can lead to needless code duplication and even non-termination (resulting from, for example, duplicating instructions in the loop header block when statically branching to it from the latch).

**Annotating Loops.** The way the specializer does loop unrolling is by cloning the body of a loop per each set of values of variables reaching its header (see Sect. 3.3). If any of such variables are static and not constants, but the loop is controlled by a dynamic condition, then the specializer won't terminate.

```

loop:
  %n = phi i32 [ 0, %0 ], [ %n.inc, %loop ], !static !0
  %n.inc = add i32 %n, 1, !static !0
  %cmp = icmp slt i32 %n, %d
  br i1 %cmp, label %loop, label %exit

```

**Fig. 3.** Static variables in the header of a dynamic loop

The problem is illustrated by Fig. 3. On each subsequent iteration of this loop, the values of static variables `%n` and `%n.inc` will be different: (0, 1), (1, 2), (2, 3) and so on, and the specializer won't terminate in an attempt to unrolling it.

The solution is to check if exit conditions in the latch blocks of a loop are dynamic, and if this is the case, annotate all changing variables in the header of the loop as dynamic.

**Annotating Function Calls.** Function calls are an exception to the congruence condition because a call can be annotated static even if some of its arguments are dynamic. The reason is that the semantics of a call instruction being static is different from that of other instructions: static calls are still performed at run time (unless the function is completely static (see Sect. 3.3) — but this is nothing but a minor optimization, and nothing is conceptually changed in what follows), the binding time simply indicates that the return value is known at specialization time, before the call has to be made at run time.

The binding time of a call instruction is hence determined by the binding times of values returned from the called function when the latter is analyzed according to the argument division at the call site. Therefore, annotating a function call requires performing binding-time analysis for a called function. On the other hand, the binding time of a call instruction may influence the binding time of a value returned from the function as well. In case of recursive function calls, this is a cyclic dependency that needs to be resolved.

The simplest way to resolve this is to initialize return binding times for all functions as dynamic and use this default value for all calls which are back edges in the call graph. This has an obvious downside of failing to handle static recursion as such — but the latter is barely used in PostgreSQL source code which is our primary application (see Sect. 4).

The algorithm that we use for analyzing a single function is thus as follows:

1. Initially, annotate all function results as dynamic and all call instructions as static.
2. Upon reaching a fixed point in annotating the individual instructions, recursively analyze all called functions that are not already being analyzed, according to the argument divisions at corresponding call sites. Reannotate call instructions as dynamic if return binding times of corresponding functions are dynamic.
3. If binding time of at least one of the calls has changed, analyze the function again until the fixed point is reached, and repeat the algorithm.
4. If no binding times have changed, change callees appropriately (to refer to the newly annotated functions) and update the binding time of the return value of this function.

Determining binding times of call instructions requires recursively analyzing called functions, perhaps multiple times with different argument divisions, generating annotated functions which won't necessarily be used in the end. This may lead to unpredictable performance of the BTA phase as a whole, although its termination is guaranteed (see below). This is not, however, a major problem for runtime specialization since BTA is expected to be performed only once.

**Termination.** Here, we briefly show that binding-time analysis as described in this section always terminates.

Since no function can be annotated twice with regards to a particular argument division and since there are only finitely many function—argument division pairs in any given source program, BTA always terminates as long as annotating a particular function with respect to a particular argument division terminates.

In the course of annotating instructions in a particular function, binding time of any particular instruction never changes from dynamic to static. For calls, this is true because binding time of a function result can't ever change from dynamic to static in case binding time of a particular argument changes from static to dynamic. Since for any particular function annotation only finitely many  $S \rightarrow D$  binding-time changes are possible, annotating a particular function always terminates.



### 3.3 Specialization

Specialization is driven by binding-time annotations produced in the BTA phase. As described here, it is polyvariant both in functions and basic blocks, meaning that for each new pair of an annotated function and a list of static argument values, a new residual function is constructed, and for each new pair of a basic block and a static store containing values of reaching definitions, a new residual basic block is constructed.

Function specialization starts from the entry basic block of a function and the initial static store containing only the values of static arguments in the function call, and proceeds by repeatedly evaluating basic blocks and their successors and updating corresponding static stores, until there are no more pairs of (block, store) to evaluate. This strategy effectively propagates particular static values towards their uses by duplicating conditional branches and unrolling loops.

Any particular annotated basic block can be evaluated multiple times, but each time a new residual basic block is constructed, a new set of residual definitions is built from the same set of dynamic instructions in the annotated code, thus maintaining the SSA property automatically.

Result of specializing function `@power.ds` (Fig. 1) with respect to `%n=4` is shown in Fig. 4.

```
define i32 @power.4(i32 %x) {
  %x.sqr = mul i32 %x, %x
  %x.sqr.1 = mul i32 %x.sqr, %x.sqr
  ret i32 %x.sqr.1
}
```

Fig. 4. Result of specialization for example in Fig. 1

**Static Store.** Static store is a data structure that maps static variable definitions to particular constant values. Each definition can have multiple values due to the polyvariance of specialization in basic blocks: each basic block, for example in a body of a static loop, can be evaluated multiple times, each time resulting in a new residual basic block.

Static store is organized as a tree of scopes. At any point during specialization process, the state of the specializer includes some particular leaf node in the store, and new leaf nodes can be added when a control flow edge is visited. Any particular leaf node is associated with a particular residual basic block, and every node on the path to the root scope corresponds to some block that dominates the current block in the CFG — in a way that every dominator has at least one scope associated with it, and each pair of basic blocks that corresponds to a pair of scopes immediately following one another in the tree, is itself connected by a control-flow edge.

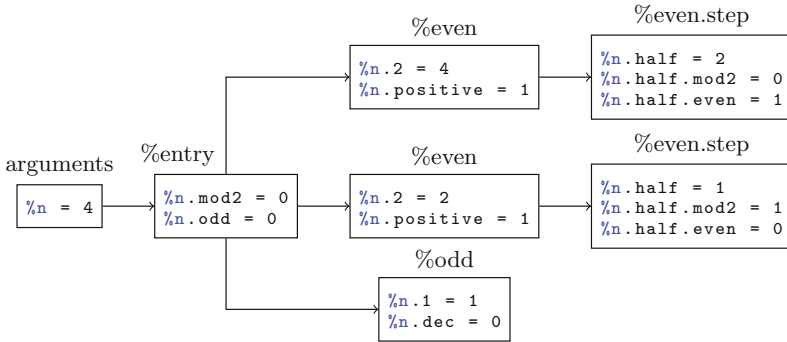


Fig. 5. Static store example

Example in Fig. 5 shows the state of the static store created during specializing the function @power.ds (Fig. 1) to %n=4 (see the residual code in Fig. 4). The root scope of the store contains the value for the static argument %n. During specialization, the basic blocks %even and %even.step are visited twice each, and so two pairs of scopes are created.

**Static Functions.** If all non-terminating instructions in a function are annotated as static, then the function represents a completely static calculation and does not need a residual function. In fact, if such a function was to be constructed at specialization time, it would contain nothing but a single return instruction with a static value.

As an optimization, these cases are annotated appropriately at binding-time analysis time, and no residual function is created at specialization time — instead, the specializer evaluates the entirety of the function statically, and replaces corresponding calls with a single static result.

## 4 Runtime Specialization of PostgreSQL Query Executor

We finally describe the application of program specialization to PostgreSQL query executor.

Query execution in PostgreSQL is implemented in several stages: parsing and rewriting, which manipulate a query’s syntactic representation, optimization, which constructs a plan tree, and execution, which interprets a plan tree (using Volcano (also known as pull-based) execution model [G94]) in order to obtain the result.

PostgreSQL provides hooks for extensions into almost all of these stages. In particular, ExecutorRun hook provides the ability to replace the default execution strategy (plan interpretation) with query compilation.

Given a particular SQL query, our specialization-based query compiler specializes PostgreSQL source code on the fly in order to obtain efficient machine code. Overall, the method can be summarized by the following:

1. At compilation time (only for offline partial evaluation):
  - (a) First, PostgreSQL source code is compiled with Clang, which results in an LLVM IR module containing a function named `ExecutePlan`, which is an entry point to the built-in query interpreter. This function takes as arguments a query plan and an execution context and evaluates the plan according to the context.
  - (b) The function `ExecutePlan` is then annotated according to its binding-time division, which results in a new function `ExecutePlanann`. All direct and indirect callees are annotated as well.
2. At run time:
  - (a) The function `ExecutePlanann` is specialized with respect to a particular query plan and execution context at hand, resulting in `ExecutePlanres`.
  - (b) The function `ExecutePlanres` is further optimized and compiled by LLVM JIT to machine code, which is then immediately executed.

Along with the query-executor-related parts of PostgreSQL source code, binding-time analysis for offline partial evaluation also takes as input the LLVM IR file with binding-time type definitions of struct types used when annotating memory operations (see Sect. 3.2). Notable examples of such struct types are query plan nodes and expression nodes.

Online partial evaluation does not have a compile-time phase, but instead requires a run-time preprocessing phase to gather all memory addresses of constants in the heap (in order to define the `heap()` function — see Sect. 2).

## 5 Conclusion

In this paper we described program specialization methods and their application to PostgreSQL query executor.

We are developing prototype query compilers based on online (Sect. 2) and offline (Sect. 3) partial evaluation. Online specializer is implemented for PostgreSQL 9.6 and shows up to 1.4x speedup on some synthetic queries. Offline specializer is implemented for PostgreSQL 10 and shows up to 1.4x speedup on TPC-H Q1 which is part of the industry-standard TPC-H benchmark [TPC-H]. Offline specializer currently requires some minor binding-time improvements applied to the PostgreSQL codebase.

Our experiments show that runtime specialization can be used to effectively eliminate interpretation overhead and inline static query and database parameters into the compiled machine code, but its performance is not currently on par with that of the traditional query compiler [MB17] that we are developing separately, which shows up to 5.5x speedup on Q1. We speculate that this difference is due to push-based execution model that the compiler implements and other algorithmic improvements that can't be automated.

We propose implementing algorithmic improvements separately in C and applying runtime specialization on top. We started by implementing the push model for PostgreSQL. Combined with the offline specializer, it shows 1.6x speedup on TPC-H Q1 compared to PostgreSQL 10.

Our implementation also suggests that query compilers can effectively combine specialization of some parts of an interpretive query engine with traditional compiled implementation of the other in order to maximize efficiency of generated code.

## References

- [CPS+81] Chamberlin, D.D., Putzolu, F., Selinger, P.G., Schkolnick, M., Slutz, D.R., Traiger, I.L., Yost, R.A.: A history and evaluation of System R. *Commun. ACM* **24**(10), 632–646 (1981). <https://doi.org/10.1145/358769.358784>
- [G94] Graefe, G.: Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.* **6**(1), 120–135 (1994). <https://doi.org/10.1109/69.273032>
- [Gre99] Greer, R.: Daytona and the fourth-generation language cymbal. *Sigmod*, 525–526 (1999). <https://doi.org/10.1145/304181.304242>
- [KVC10] Krikellas, K., Viglas, S.D., Cintra, M.: Generating code for holistic query evaluation. In: *Proceedings of International Conference on Data Engineering*, pp. 613–624 (2010). <https://doi.org/10.1109/ICDE.2010.5447892>
- [LC14] Lomuller, V., Charles, H.-P.: A LLVM extension for the generation of low overhead runtime program specializer. In: *Proceedings of International Workshop on Adaptive Self-Tuning Computing Systems - ADAPT 2014*, pp. 14–16 (2014). <https://doi.org/10.1145/2553062.2553064>
- [LLVM] The LLVM Compiler Infrastructure. <http://llvm.org/>
- [MB17] Melnik, D., Buchatskiy, R., Zhuykov, R., Sharygin, E.: JIT-compiling SQL queries in PostgreSQL using LLVM. Presented at PGCon 2017 (2017). <http://www.pgcon.org/2017/schedule/events/1092.en.html>
- [Neu11] Neumann, T.: Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.* **4**(9), 539–550 (2011). <https://doi.org/10.14778/2002938.2002940>
- [NL14] Neumann, T., Leis, V.: Compiling database queries into machine code. *IEEE Data Eng. Bull.* **37**(1), 3–11 (2014)
- [Pos] PostgreSQL Global Development Group. PostgreSQL. <http://www.postgresql.org/>
- [RO10] Rompf, T., Odersky, M.: Lightweight modular staging. In: *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering - GPCE 2010*, p. 127 (2010). <https://doi.org/10.1145/1868294.1868314>
- [Smo14] Smowton, C.S.F.: I/O Optimisation and elimination via partial evaluation. University of Cambridge, Computer Laboratory, Ph.D. thesis, (UCAM-CL-TR-865), 1131 (2014)
- [TPC-H] TPC-H, an ad-hoc, decision support benchmark. Transaction Processing Performance Council. <http://www.tpc.org/tpch>
- [WWS+12] Wurthinger, T., Wob, A., Stadler, L., Duboscq, G., Simon, D., Wimmer, C.: Self-Optimizing AST Interpreters (2012)
- [ZDS12] Zhang, R., Debray, S., Snodgrass, R.T.: Micro-specialization: dynamic code specialization of database management systems. In: *International Symposium on Code 6373* (2012). <https://doi.org/10.1145/2259016.2259025>

- [ZSD12a] Zhang, R., Snodgrass, R.T., Debray, S.: Micro-specialization in DBMSes. In: Proceedings - International Conference on Data Engineering, pp. 690–701 (2012). <https://doi.org/10.1109/ICDE.2012.110>
- [ZSD12b] Zhang, R., Snodgrass, R.T., Debray, S.: Application of micro-specialization to query evaluation operators. In: Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012, pp. 315–321 (2012). <https://doi.org/10.1109/ICDEW.2012.43>

# MicroTESK: A Tool for Constrained Random Test Program Generation for Microprocessors

Alexander Kamkin<sup>1,2,3,4</sup> and Andrei Tatarnikov<sup>1,4</sup>(✉)

<sup>1</sup> Institute for System Programming of the Russian Academy of Sciences,  
Moscow, Russia

{kamkin, andrewt}@ispras.ru

<sup>2</sup> Lomonosov Moscow State University, Moscow, Russia

<sup>3</sup> Moscow Institute of Physics and Technology,  
Dolgoprudny, Moscow Region, Russia

<sup>4</sup> National Research University Higher School of Economics, Moscow, Russia

**Abstract.** The paper presents MicroTESK, a tool for test program generation for functional verification of microprocessors. It generates test programs from templates which describe generation tasks in terms of constraints that must be satisfied in order to reach certain coverage goals. The tool uses formal specifications of the instruction set as a source of knowledge about the microprocessor under verification. This gives several advantages. First, the tool is easily adapted to new architectures by providing corresponding specifications. Second, constraints that constitute coverage model are automatically extracted from specifications. Third, a reference model used to track the microprocessor state during test generation is constructed on the basis of specifications. Such an approach helps to reduce the effort required to create test programs and increase the quality of testing. The tool has been successfully applied in industrial projects for verification of ARMv8 and MIPS64 microprocessors.

**Keywords:** Microprocessors · Functional verification  
Test program generation · Formal specifications  
Instruction set architectures

## 1 Introduction

*Functional verification* is an integral part of the microprocessor design process. Its task is to ensure that the implementation being developed conforms to the specification. Growing complexity of modern microprocessors makes this task extremely challenging. According to various estimates, functional verification accounts for up to 80% of overall project resources.

In current industrial practice, functional verification mainly relies on simulation-based techniques [1]. They imply executing stimuli on the design prototype and comparing its behavior with the expected behavior described by the specification. Stimuli used for microprocessor verification are usually represented by streams of instructions, which constitute *test programs*. This approach

is the most natural since *instruction set architecture* (ISA) is the only interface available to users to interact with a microprocessor.

Test programs are created according to verification requirements with the help of special automation tools referred to as *test program generators* (TPGs) or *instruction stream generators* (ISGs). Verification requirements are formulated in the *verification plan* that enumerates *test situations* to be covered. Normally, it is described in a natural language and derived from the design specifications. The task of translating these descriptions into test programs is not straightforward since requirements can be quite complex. They can cover situations related to pipelining, memory management and multicore execution. In addition, constructed tests must comply with the validity requirements imposed by the ISA.

TPGs based on pseudorandom generation, which are widely used in practice, are not capable of covering verification requirements in a systematic way. Stimuli generated by such tools are often prone to redundancy and insufficient coverage. One of the possible solutions to this problem is to utilize constraint solving techniques. In this approach, a TPG constructs stimuli by solving *constraint satisfaction problems* (CSPs) that correspond to verification requirements. However, due to complexity, constraints may not take into account all factors that determine the behavior of the microprocessor. Therefore, to increase chances of hitting “interesting” situations, constraint solving is combined with randomization. Such an approach is known as *constrained random generation*.

One of the tools implementing this approach is MicroTESK [2] developed at ISP RAS. The key feature of MicroTESK is using *formal specifications* as a source of knowledge about the *design under verification* (DUV). Information provided in specifications is used in three ways: (1) to get assembly format of instructions; (2) to build constraints corresponding to various test situations; (3) to construct a *reference model* of the DUV. The reference model is an *instruction set simulator* (ISS) which is used to track the state of the DUV during test generation. It helps to maintain a context for constraint solving and to ensure validity of the generated stimuli. Generation tasks are formulated in the form of *test templates* in a Ruby-based language, which specify instructions to be used, their order and constraints on their operands. The approach facilitates creating high-quality tests and simplifies configuring the TPG for testing new designs.

The rest of the paper is organized as follows. Section 2 gives a brief overview of existing tools for constrained random generation. Section 3 describes the approach used by MicroTESK. Section 4 contains case studies of applying MicroTESK in industrial projects. Section 5 concludes the paper.

## 2 Related Works

Constrained random generation has been applied for functional verification of microprocessors since the end of the 1990s. Despite significant effort put into development of industrial TPGs, none of them can be considered as a universal solution to all verification tasks. They have different characteristics and application area. For example, they may support different ISAs, use different constraint-solving methods and have different degree of randomness and generation speed.

The best known TPG is Genesys-Pro by IBM Research [1]. It generates random and constraint-based tests from two types of input data: (1) *microprocessor model* and (2) *test templates*. The first provides a declarative description of the DUV's ISA which includes signatures of instructions, their semantics in the form of CSPs and related heuristics that help increase coverage. The second represents an abstract description of test scenarios, where situations to be covered are specified in terms of CSPs and heuristics defined by the microprocessor model. Genesys-Pro performs generation in an instruction-wise manner: at each step, it selects an instruction, solves CSPs for the chosen instruction, simulates the instruction, and, finally, prints it in the assembly format. Microprocessor models are described in XML using special building blocks. Test templates are created in a special domain-specific language, which provides constructs for describing generation tasks. To simulate instructions, the tool uses an external ISS integrated with the help of special libraries. Genesys-Pro lacks support for modeling floating-point and memory access instructions. To create tests for such instructions, additional tools are used. They provide facilities to specify and solve specialized CSPs for corresponding test situations.

Another well-known TPG is RAVEN (Random Architecture Verification Machine) [3] developed by Obsidian Software, which is now owned by ARM. Like Genesys-Pro, it uses microprocessor models to specify the DUV configuration and test templates to describe test scenarios. Models are described in XML, while test templates are created in a domain-specific language. Test templates are focused on coverage grids and use CSPs to formulate specific coverage goals. RAVEN simulates generated instructions in an external ISS integrated into the tool. As we can see, RAVEN is based on the same principles as Genesys-Pro, but it is more oriented on random generation and uses CSPs to shift bias towards certain areas. Also, there are differences in formats of input data, which affect usability.

Common issues of these TPGs are difficulties related to configuring them for a new DUV: creating descriptions of the model in multiple formats, which might not be easy to understand, and integrating external ISSs. So, this is likely to require close interaction with TPG developers. An important point is that CSP descriptions have to be created by hand. This is laborious and imposes a risk that some test situations will be missed.

### 3 MicroTESK Approach

#### 3.1 Key Requirements

To provide a high level of coverage, a TPG requires information on the DUV. Since the verification plan is based on the design specifications, it would be logical to use descriptions derived from these specifications. Having such descriptions in a standardized format understandable to verification engineers would simplify the maintenance of the TPG. There is a family of formal languages known as *architecture description languages* (ADLs) [4] that would be suitable for this task. Such languages have been actively used to create disassemblers, ISSs and



retargetable compilers. In addition, ADL specifications can be used to extract constraints.

Another issue to consider is the *test template description language*. This language must be easy to learn for verification engineers. It must allow describing generation tasks for any ISA and establishing constructs to describe new types of generation tasks. Also, since modern industrial testbenches can contain thousands lines of code, it must provide facilities to organize template code into reusable libraries. It would be reasonable to take advantage of a well-trying high-level language extended with domain-specific constructs. This would give the language power and flexibility and decrease the learning effort.

An important part of TPG functionality is construction of instruction sequences. To exercise the DUV behavior in complex situations that require large sets of CSPs to be satisfied, it is efficient to create stimuli by merging smaller instruction sequences with the help of random or combinatorial algorithms. A TPG must provide such facilities. To accommodate various verification requirements, the set of supported merging strategies must be flexible and allow extension.

Finally, a TPG must support various constraint solving methods. Constraints applied to stimuli include: (1) constraints on memory location; (2) constraints on execution path of specific instructions; (3) constraints related to control flow of the program; (4) floating-point constraints; (5) MMU-related constraints; (6) pipeline-related constraints. They are solved with different engines, which must be integrated into the TPG to be used in combination.

## 3.2 MicroTESK Architecture

MicroTESK is divided into two main parts: (1) the *modeling framework* and (2) the *testing framework*. The first one processes formal specification to construct a microprocessor model that holds all design-specific information. The second one generates stimuli on the basis of the model and templates provided by users. The architecture of MicroTESK is shown in Fig. 1. The model consists of the following components: (1) the *metadata* that provides a catalogue of supported instructions, their arguments and associated test situations; (2) the ISS that simulates execution of instructions; (3) the *coverage model* that holds constraints extracted from formal specifications.

## 3.3 Modeling Framework

The modeling framework includes a set of translators that analyze formal specifications, extract the necessary information and construct the model using special libraries.

MicroTESK uses ISA specifications created in the nML ADL [5]. They include descriptions of constants, data types, registers, addressing modes, instructions, memory and temporary variables. The syntax of nML is very close to notations used in microprocessor architecture manuals to describe instruction semantics. For example, here is a description of the ADD instruction from the MIPS64 manual:

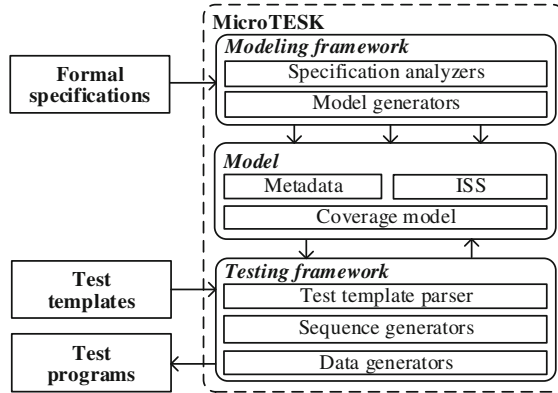


Fig. 1. Architecture of the MicroTESK TPG

```

if NotWordValue(GPR[rs]) or NotWordValue(GPR[rt]) then
  UNPREDICTABLE
endif
temp → GPR[rs]31 || GPR[rs]31..0 + (GPR[rt]31 || GPR[rt]31..0)
if temp32 ≠ temp31 then
  SignalException(IntegerOverflow)
else
  GPR[rd] → sign_extend(temp31..0)
endif

```

Such descriptions can be turned into formal specifications with minimal effort. The code below shows how the ADD instruction and resources it accesses can be specified in nML:

```

type DWORD = card(64)
reg GPR [32, DWORD]
mode R (i: card(5)) = GPR[i]
  syntax = format("%d", i)
  image = format("%5s", i)

var temp33[card(33)]
op add (rd: R, rs: R, rt: R)
  syntax = format("add %s, %s, %s", rd.syntax, rs.syntax, rt.syntax)
  image = format("000000%5s%5s%5s00000100000", rs.image, rt.image, rd.image)
  action = {
    if sign_extend(WORD, rs<31>) != rs<63..32> || sign_extend(WORD, rt<31>) != rt<63..32> then
      unpredicted;
    endif;
    temp33 = rs<31>::rs<31..0> + rt<31>::rt<31..0>;
    if temp33<32> != temp33<31> then
      exception("IntegerOverflow");
    else
      rd = sign_extend(DWORD, temp33<31..0>);
    endif;
  }
}

```

The configuration of MMU is specified using a specialized language called MMUSL, which extends nML with facilities to describe address types, segments, buffers, tables and logic of memory accesses. More details and examples are provided in works [6, 7].

### 3.4 Testing Framework

The role of the testing framework is to generate test programs by processing test templates. To create test templates, a specialized Ruby-based language [8] is used. Templates are represented by classes derived from a special library class which provides constructs to describe generation tasks. Architecture-specific constructs are added dynamically using Ruby’s metaprogramming facilities.

Instruction streams described by templates consist of the following parts: (1) a prologue that initializes the environment, (2) stimuli that perform specific actions and check the results, (3) dispatching code that performs control transfers between different parts and (4) an epilogue that terminates the execution. Since different parts solve different tasks and might be based on different generation methods, they are generated and simulated separately. When it is required to simulate sequences produced by different template parts as a single scenario, dispatching code is used. For example, different parts can be executed on different processing elements of a multiprocessor.

The generation process includes the following stages: (1) parsing the template; (2) constructing instruction sequences; (3) solving CSPs to generate data and constructing initialization code to assign the data values to input arguments; (4) executing the instruction sequences in the ISS; (5) creating self-checks based on information provided by the ISS; (6) printing the resulting instruction sequences to a file.

For example, below is a test template that describes all possible pairs of ADD and SUB instructions with applied constraints “Normal” and “IntegerOverflow”.

```
class MyTemplate < Template
  def initialize ... end
  def pre      ... end
  def post     ... end
  def run
    block(:combinator => 'product') {
      iterate {
        add t0, t1, t2 do situation('Normal') end
        add t0, t1, t2 do situation('IntegerOverflow') end
      }
      iterate {
        sub t3, t4, t5 do situation('Normal') end
        sub t3, t4, t5 do situation('IntegerOverflow') end
      }
    }.run
  end
end
```

CSPs are solved with generic SMT-solvers like Z3 and CVC4 or with specialized engines. The data generation engine facilitates adding support for new solvers.

## 4 Practical Application

MicroTESK has been applied for verification of ARMv8 and MIPS64 microprocessors. Also, there is an ongoing research dedicated to verification of PowerPC and RISC-V microprocessors. Table 1 provides metrics for the created specifications. The average generation speed of the MicroTESK is about 3500 instructions/second for random generation and 100 instructions/second for constrained generation.

**Table 1.** Size of formal specifications and required effort

Project	ARMv8	MIPS64	PowerPC	RISC-V
Number of instructions	795	220	34	63
Size of ISA specifications (lines of code)	12220	3999	935	816
Size of MMU specifications (lines of code)	2119	267	0	0
Efforts (person-months)	13	4	1	0.75

## 5 Conclusion

In the paper the MicroTESK [2] TPG was presented. MicroTESK has been successfully applied in industrial projects for verification of ARMv8 [6] and MIPS64 [9] microprocessors.

## References

1. Adir, A., Almog, E., Fournier, L., Marcus, E., Rimon, M., Vinov, M., Ziv, A.: Genesys-pro: innovations in test program generation for functional processor verification. *Des. Test Comput.* **21**, 84–93 (2004)
2. MicroTESK page. <http://forge.ispras.ru/projects/microtesk>
3. RAVEN test program generator. <http://www.slideshare.net/DVClub/introducing-obsidian-software-andravengcs-for-powerpc>
4. Mishra, P., Dutt, N. (eds.): *Processor Description Languages. Systems on Silicon*. Morgan Kaufmann, San Francisco (2008). 432 pages
5. Freericks, M.: *The nML machine description formalism*. Technical report TR SM-IMP/DIST/08, TU Berlin CS Department (1993)
6. Chupilko, M., Kamkin, A., Kotsynyak, A., Protsenko, A., Smolov, S., Tatarnikov, A.: Specification-based test program generation for ARM VMSAv8-64 memory management units. In: *Workshop on Microprocessor Test and Verification*, pp. 1–6 (2015). <https://doi.org/10.1109/MTV.2015.13>
7. Kamkin, A., Kotsynyak, A.: Specification-based test program generation for MIPS64 memory management units. In: *Trudy ISP RAN [Proceedings of ISP RAS]*, vol. 28, part 4, pp. 99–114 (2016)
8. Tatarnikov, A.: Language for describing templates for test program generation for microprocessors. In: *Trudy ISP RAN [Proceedings of ISP RAS]*, vol. 28, part 4, pp. 81–102 (2016)
9. MicroTESK for MIPS64 page. <http://forge.ispras.ru/projects/microtesk-mips64>

# Enriching Textual Xtext-DSLs with a Graphical GEF-Based Editor

Marcel Toussaint and Thomas Baar<sup>(✉)</sup>

Hochschule für Technik und Wirtschaft (HTW) Berlin,  
Wilhelminenhofstraße 75A, 12459 Berlin, Germany  
m.toussaint@web.de, thomas.baar@htw-berlin.de

**Abstract.** Xtext is a widely accepted framework to develop domain-specific languages (DSLs). However, these DSLs are bound to be purely textual, what is appropriate in many but not all cases. Sometimes, one wishes to have another concrete syntax for a DSL. For example, a model should be represented only by graphical elements (i.e., a purely graphical syntax) or by a mixture of graphical elements and textual annotations (i.e., a hybrid syntax).

In this paper, we describe an approach for developing a graphical editor for a hybrid concrete syntax of a given DSL. The starting point is an Xtext grammar of a DSL, together with all the usual accompanying features such as validators and code generators. We describe the necessary steps to enrich the existing toolset with a graphical editor based on the GEF framework.

Our approach is highly flexible since large parts of the editor can be implemented and tailored manually to accommodate the underlying DSL. Nevertheless, the effort to create such an editor is manageable, since the GEF framework offers most of the necessary features for graphical editing. We describe what conditions must be met with regards to the underlying language designed with Xtext and how a corresponding graphical GEF-based editor for a hybrid syntax can be implemented.

**Keywords:** Xtext · Graphical syntax · GEF · DSL

## 1 Introduction

Over the last decade, there has been an increased interest in domain-specific languages (DSLs) in academic research [2]. Domain-specific languages are specialized computer languages that are created to express problems and solutions in a specific problem domain. Compared to general purpose languages, DSLs allow for an expression using the same level of abstraction as is commonly used in the particular problem domain. This means, that domain-specialists should be able to use a DSL to describe certain problems in their domain in an easily understandable way without extensive additional knowledge on programming languages [6].

There is a multitude of frameworks and tools like the *JetBrains Meta Programming System* [9], the *MetaEdit+ Domain-Specific Modeling (DSM) environment* [10], or the *Spoofax Language Workbench* [13] available at this time to implement domain-specific languages. In most cases, one would implement a language by defining a meta-model containing all the necessary modeling concepts together with their attributes. This language meta model builds the foundation of the language to model the desired aspects. Instances of this language (i.e., models) are often developed in a purely textual way. There are, however, situations where a graphical representation of such models and especially the ability to edit such models via a graphical editor would be useful and desirable. To address this, the *Graphical Modeling Project (GMP)* sought to provide generic mechanisms to outfit textual syntax with graphical syntax. This generic approach turned out to be rather elaborate [12]. Although literature reports on a robust approach for synchronous editing of the textual and the graphical representation of a model in the context of the aforementioned *Spoofax* [11], this approach cannot yet be considered mainstream. In this paper we propose a solution to implement a tailored editor application for a graphical model representation (i.e., graphical or hybrid syntax) for a specific user-defined DSL. The DSL has been defined with the *Xtext Framework* [7]. We will showcase a workflow to (a) identify the requirements in regards to the underlying DSL, (b) to choose, if applicable and useful, appropriate graphical representations of the entities defined in this DSL and (c) to synchronize the textual to the graphical model and vice versa.

The rest of the paper is organized as follows: In Sect. 2 we describe the utilized *Xtext* and *GEF* frameworks together with a simple example of an Xtext-DSL to model state machines in a textual manner. A showcase model of said language will be described as a textual Xtext representation. In Sect. 3 we propose a potential solution for implementing an editor application using the GEF framework to create and manipulate textual Xtext models via their graphical representation. In Sect. 4 we describe the subsumption of this project into current related works and subsequently in Sect. 5 we outline potential future extensions of this project considering the synchronization mechanisms developed and the lessons learned so far.

## 2 Basic Concepts

### 2.1 The Open Source Framework Xtext

Xtext is an open-source framework for the development of domain-specific languages. Xtext is developed in the *Eclipse Project* as part of the *Eclipse Modeling Framework*. Compared to other frameworks for implementing DSLs like the aforementioned *JetBrains Meta Programming System* or the *MetaEdit+ Domain-Specific Modeling (DSM) environment*, Xtext offers the advantage of a complete integration into the Eclipse IDE. It allows the usage of typical editor features such as syntax highlighting, code completion and error markers, when editing a model in the defined DSL. The models of the created language

are purely textual. Typically, a language environment also includes user-defined code generators to generate code of a target language from the edited DSL model. The (abstract) syntax of DSLs defined with Xtext can be of arbitrary complexity since every valid syntax tree must pass all — possibly sophisticated — syntax checks defined by so-called validators.

```

8=Stateachine:
9  'stateachine' name=ID
10 'variables:' variables+=Variable (',' variables+=Variable)*
11 'events:' events+=Event (',' events+=Event)*
12 'states:' states+=State (',' states+=State)*
13 'transitions:' (transitions+=Transition*);
-----
1 stateachine parkingTicketMachine
2 variables: collected INT, bill INT
3 events: always, cardInserted, coinInserted, shutdown
4 states: Stop, Start, Idle, WaitingForMoney, Paid
5 transitions:
6   Start to Idle when always / bill=5, collected=0;
7   Idle to WaitingForMoney when cardInserted;
8   WaitingForMoney to WaitingForMoney when coinInserted [collected<bill-1] / collected +=1;
9   WaitingForMoney to Paid when coinInserted [collected==bill-1] / collected +=1;
10  Paid to Idle when always / collected=0;
11  Idle to Stop when shutdown;

```

**Fig. 1.** Grammar of a DSL to model simple finite-state machines and valid model (excerpt)

Figure 1 shows an excerpt from the grammar of an Xtext language for simple finite-state machines (upper part) and a valid (i.e. correct w.r.t. the language syntax) textual model (lower part) that will be used as a running example. Each model expressed in the language represents a self-contained state machine with states, transitions, events and optional constraints for state transitions in form of simple Boolean expressions on defined variables.

## 2.2 The Graphical Editing Framework (GEF)

The Graphical Editing Framework GEF [3] provides end user tools integrated into the Eclipse IDE as well as components to create rich JAVA editor applications. GEF offers a great number of prefabricated features to implement graphical editors like mechanisms to deal with multi-selects, drag and drop functionalities and undo/redo mechanisms.

Figure 2 shows the basic structure of a GEF editor. GEF uses a Model-View-Controller concept to compartmentalize the graphical representation and the underlying semantic model. Each model (in GEF terminology *content model*) usually consists of instances of different types representing the individual entities in the model (i.e. in case of the aforementioned state machine model, states, transitions etc.). For each such instance type a controller (*model part*) has to be implemented to connect the model instance with its graphical representation (*Visual*). By providing interfaces and adapters for different graphical representations like *JavaFX* or *SWT*, GEF offers a great level of exchangeability regarding the actual UI-representation.

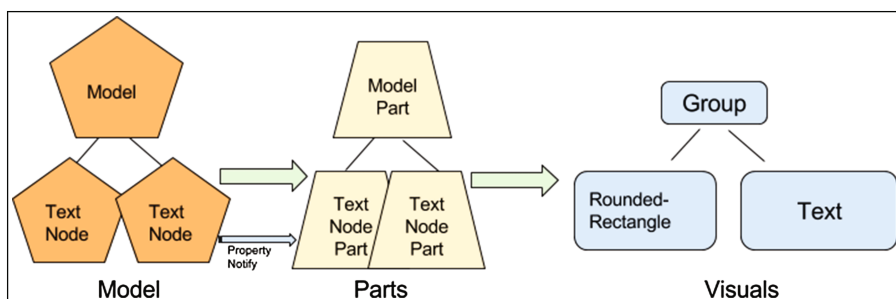


Fig. 2. GEF MVC architecture [1]

### 3 Implementing a GEF-Based Editor to Create DSL Models

In the following subsections we describe our approach and lessons learned while implementing a graphical GEF-based editor application for our example DSL<sup>1</sup>.

#### 3.1 Architecture of the Editor Application

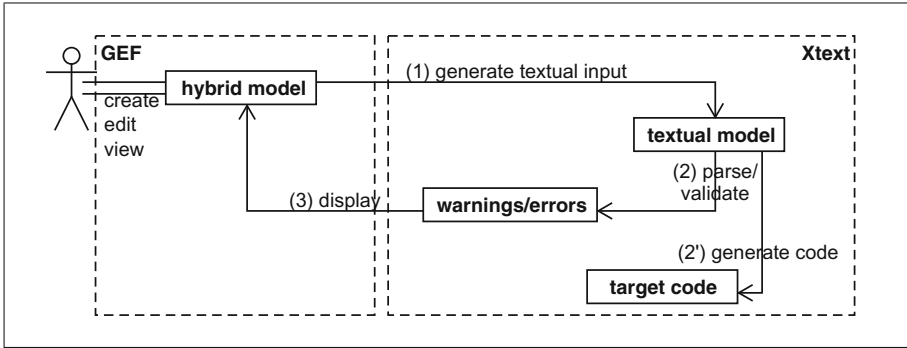
The starting point for the editor application is the aforementioned Xtext DSL to model finite-state machines. Consider you have been given this DSL grammar accompanied by some custom syntax validators for the textual models of that language. Using the onboard features of the Xtext framework, creating models in that Xtext language is only possible in a purely textual manner. To create an editable graphical representation for these models, a graphical syntax has to be found that maps each modeling concept from the textual DSL grammar.

For some modeling concepts it might not be a simple task to find meaningful graphical equivalents. Therefore a hybrid syntax might prove more useful compared to a purely graphical syntax. In regards to our state machine language this means that while for example a graphical representation for transition constraints (which are represented as Boolean expressions in the grammar) can certainly be found, the textual expressions might be more convenient to use even in the context of a graphical editor. Models created with the help of the hybrid syntax have to be parsed against the grammar of the language for validation. To do so, it is necessary to transform the hybrid model into an equivalent textual model that can be parsed using the provided validators. These considerations led us to the following model of the principle workflow for the graphical model editor.

Figure 3 shows the principle workflow of the editor application. Models can be created and edited by the user utilizing the hybrid syntax provided. When requested by the user, the emerging model is transferred into a corresponding

<sup>1</sup> The described editor together with its source code is freely available from <https://github.com/m-toussaint/SME>.





**Fig. 3.** Graphical editor workflow

textual model (1) which can then be parsed against the underlying DSL by using the generated parser and the implemented validators (2). If the model is parsed without errors, additional Xtext features like the code generation can be triggered (2'). If the textual representation of the editor model however cannot be parsed without errors, the emerging error messages are masked and displayed in the graphical editor application (3).

With the help of the created editor application based on the presented workflow we are able to create editable hybrid representations of our underlying textual DSL. Figure 4 shows the graphical/hybrid representation of the example state machine illustrated in Sect. 2 in the implemented GEF editor application.

### 3.2 Utilizing AST Classes as Content Model

Since Xtext is based on the *Eclipse Modeling Framework (EMF)* and thus uses an *Ecore* model for the specified language, it seems obvious to use the classes of the Ecore model as content model classes for the GEF editor. This approach is certainly useful when the underlying language only consists of modeling concepts that can be purely represented with a graphical syntax. However, in case of a hybrid syntax as chosen for our running example (see Fig. 4), things are different. Though modeling concepts for states, events and variables could be straightforward derived from the Ecore model classes, the transition constraints are annotated as pure text. For these cases, the corresponding content model classes have to be implemented manually.

### 3.3 Synchronizing the Textual DSL Model and the Graphical GEF Model

Since we already store all the necessary information required for the textual representation of a graphical model in the instances of the content model classes, the synchronization from the graphical to the textual representation is rather

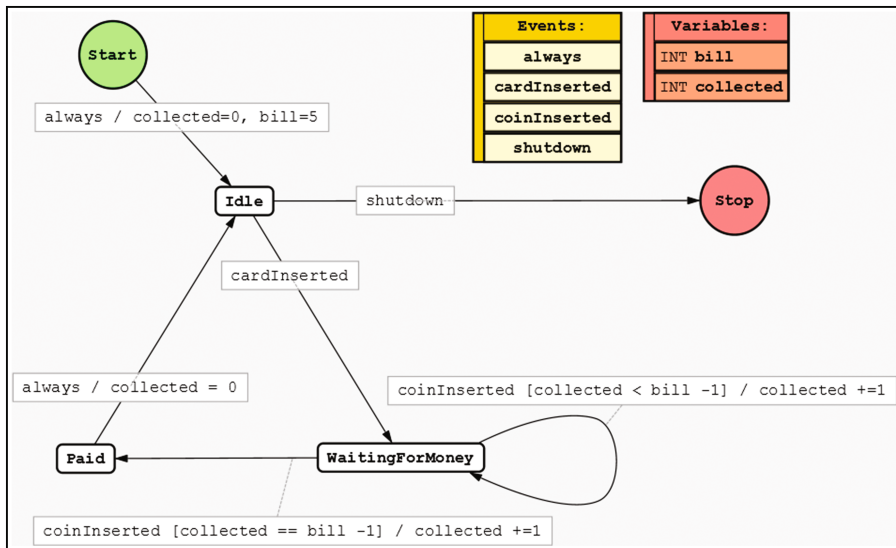


Fig. 4. Example graphical state machine model

straightforward. By extracting the required attributes, we can derive the textual model using a pretty-printing mechanism. For the most cases, the graphical model will contain more information than necessary for the textual representation (e.g. information on the position in the drawing area). However, there might be instances, where the graphical representation is actually missing required information for the textual representation. For example, the aforementioned DSL for state machines requires unique identifiers for states, but in the context of the graphical representation, states might only be represented by geometric figures without a label or a name (i.e., an identifier). When deriving the textual model these missing information have to be complemented with meaningful substitutes (e.g. automatically created IDs).

Synchronizing from the textual to the graphical representation requires more effort than the opposite direction. As we have already established, the information stored in the content model classes of the graphical editor potentially contains more information than needed for the textual model (i.e., position values, etc.). Here, we are forced to assume meaningful default values for the missing information. Take the described example DSL for state machines: each state is solely described by a unique identifier (i.e., its name or label). In the context of the graphical model, we need additional information on its position in the drawing area. For a graphical syntax, for which the position of its elements does not have any semantic significance, an auto-layout mechanism can be used to assume the missing position values.

## 4 Related Work

There is already a manifold of tools available to edit graphical syntax via an editor like for example the *YAKINDU Statechart Tools* [8]. However, these tools are bound to provide editor features for a specific syntax only (state machines in case of YAKINDU for example). Our approach describes a workflow, how to find a graphical syntax and implement a graphical editor application to create and edit models of *arbitrary DSLs*.

The project GMF [4] has the same goal as we do which is providing generic graphical representations for DSLs. However, using GMF the user is bound to develop models representing the mapping of DSL modeling concepts to their graphical representation. In contrast, our approach relies on implementing editor applications using the GEF framework. While the GMF approach sounds promising, the mapping models turned out to be quite complicated depending on the complexity of the model concepts [12].

Another interesting approach with similar goals as our's is the Eclipse project Sirius [5]. Sirius offers to define individual representations of arbitrary EMF models. Out of the box, representations in form of diagrams, tables (property views), and trees are supported. By using a pre-defined (or even user-defined) query language, one can programmatically create and manipulate the model. In case of manipulation of the underlying EMF model, Sirius updates the defined representations (graphical, tabular, or hierarchical) automatically. This difference to our approach is the starting point of the language definition: While Sirius relies on an explicit metamodel, we start with the Xtext definition of the language, i.e. with a grammar and a set of validators. This allows us to support hybrid concrete syntaxes, where parts of the model are still represented by pure text, since the mapping to the EMF classes is done in our workflow by the Xtext parser.

## 5 Conclusion and Future Works

The objective of the presented approach is to show how the process of editing DSL models of a domain-specific language can be improved by providing a graphical editor application. Although large parts of the implementation code of the editor application have to be implemented manually, the usage of the GEF framework reduces the effort to a bearable level. All necessary artifacts can be customized to accommodate preferences and characteristics of the underlying language. The advantage of substituting purely textual representations with graphical representations (at least partially) can improve in many cases the user acceptance of the language.

The editor application presented in this paper has been created under the premise, that the usage of the graphical editor should not necessarily require knowledge on the underlying DSL grammar. Therefore, the textual model representation and changes therein are not propagated to the user of the editor application. To achieve a higher level of usability and integration in the existing

tool chain provided by the Xtext framework, it would be preferable to simultaneously provide the ability to work with the graphical and the textual representation in a synchronized editor view. This is going to be the focus of our work efforts in the near future since it potentially enables users to choose which model representation they use to create and edit the model and thus alleviating the workflow. Since the Xtext framework is already seamlessly integrated into the Eclipse IDE and GEF provides the components necessary to create editor views integrated into Eclipse, other future goals would be (i) to recreate the prototype application as an Eclipse plugin, (ii) to use incremental parsing as the underlying mechanism to synchronize graphical and internal representations of the model, and (iii) to extend our graphical editor by features like code-completion.

## References

1. Benoit, F.: GEF4 tutorial. <http://fbenoit.blogspot.de/2015/11/gef4-tutorial-part-3-model-tree-and.html>
2. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Not.* **35**(6), 26–36 (2000)
3. Eclipse: GEF homepage. <https://eclipse.org/gef/>
4. Eclipse: GMP homepage. <http://www.eclipse.org/modeling/gmp/>
5. Eclipse: Sirius homepage. <https://www.eclipse.org/sirius/>
6. Hudak, P.: Modular domain specific languages and tools. In: *Proceedings of International Conference on Software Reuse (ICSR)*, pp. 134–142. IEEE (1998)
7. Itemis: Xtext homepage. <http://www.eclipse.org/Xtext>
8. Itemis: Yakindu homepage. <https://www.itemis.com/en/yakindu/statechart-tools/>
9. JetBrains: Meta programming system homepage. <https://www.jetbrains.com/mps/>
10. MetaCase: Metaedit+ domain-specific modeling homepage. <https://www.metacase.com>
11. van Rest, O., Wachsmuth, G., Steel, J.R.H., Süß, J.G., Visser, E.: Robust real-time synchronization between textual and graphical editors. In: Duddy, K., Kappel, G. (eds.) *ICMT 2013. LNCS*, vol. 7909, pp. 92–107. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-38883-5\\_11](https://doi.org/10.1007/978-3-642-38883-5_11)
12. Seehusen, F., Stølen, K.: An evaluation of the graphical modeling framework (GMF) based on the development of the CORAS tool. In: Cabot, J., Visser, E. (eds.) *ICMT 2011. LNCS*, vol. 6707, pp. 152–166. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-21732-6\\_11](https://doi.org/10.1007/978-3-642-21732-6_11)
13. Visser, E.: Spoofox language workbench homepage. [www.metaborg.org/](http://www.metaborg.org/)

# Towards Automated Static Verification of GNU C Programs

Evgeny Novikov and Ilja Zakharov<sup>(✉)</sup>

Institute for System Programming of the Russian Academy of Sciences,  
Moscow, Russia  
{novikov,ilja.zakharov}@ispras.ru

**Abstract.** Static verification based on such methods as Bounded Model Checking and Counterexample-Guided Abstraction Refinement aims at non-interactive formal proving of programs correctness against safety property specifications. To leverage existing tools for verification of a program one should prepare verification tasks first. In addition to a program fragment of a moderate size, each verification task has to contain a rather accurate model of its environment. To achieve high-quality results this model should be incrementally refined in accordance with checked safety properties. For verification of specific software, like Windows or Linux drivers, a few frameworks provide a convenient user interface and perform in an automated way generation of verification tasks, execution of static verification tools and preliminary processing of results. This paper presents a method for automated static verification of any program developed in the GNU C programming language and addresses the ongoing development of the Klever framework.

**Keywords:** Static verification · Software verification  
Formal specification · Environment model · GNU C

## 1 Introduction

Static verification allows both finding of specified safety property violations and proving of formal program correctness under certain assumptions. There are many advanced static verification tools like SLAM [1], CBMC [2], CPAchecker [3] implementing Bounded Model Checking [4], Counterexample-Guided Abstraction Refinement [5] and other methods. However, their practical application is rather limited. Despite many merits of static verification users can not leverage the tools out of the box for industrial software. Existing static verification frameworks dramatically simplify the workflow, but their scope is bound to particular kinds of programs.

We propose a new method for automated static verification of programs developed in the GNU C programming language. Currently, the method has been partially implemented within the Klever static verification framework.

---

The reported study was partially supported by RFBR, research project No. 16-31-60097.

## 2 Static Verification Background

In this section we consider an interface, features and requirements of static verification tools from the perspective of users.

Though first static verification tools appeared at the beginning of the century, their developers have been forming an active solid community. One of the most important steps in this direction was organization of a series of annual competitions on software verification, SV-COMP<sup>1</sup>. SV-COMP 2012 attracted about a dozen of static verification tool developer teams from leading research centers from all over the world [6]. The number of participants steadily grows and already 32 teams participated in SV-COMP 2017 [7]. SV-COMP competition rules de facto introduce a standard static verification tools interface accepted by all participants<sup>2</sup>.

### 2.1 Target Programs

In this paper we consider primarily static verification of programs developed in the GNU C programming language. C programs verification is of great importance since there is much such software that operates in critical domains. For instance, various operating system kernels and libraries, programming language compilers and interpreters, database systems and web servers belong to this class.

Static verification tools analyze a model of a given program extracted automatically directly from its source code. To check software developed in various programming languages tools translate programs into internal representations using appropriate front-ends first. For instance, having a program LLVM internal representation, SMACK translates it into Boogie to run static verification tools like Corral then [8]. This potentially enables support for programming languages such as C and Java, since there are appropriate LLVM front-ends. The CPAchecker static verification tool has an internal representation suitable for checking C and Java programs [3]. For parsing C programs it uses the Eclipse CDT parser.

There is the SV-COMP benchmark suite to estimate and to compare capabilities of static verification tools comprehensively. According to the competition rules this suit contains so-called verification tasks. A *verification task* contains a program and a safety properties specification. The safety properties specification represents requirements to check and we discuss this in the next subsection.

The program should be already prepared in advance so that static verification tools can take it as input without any additional processing and check it non-interactively. Programs developed in the GNU C programming language constitute the lion share of the competition benchmark suit. Thus, many participating in the SV-COMP competition static verification tools have a high-quality support of GNU C programs. Each program of the SV-COMP benchmark suite

<sup>1</sup> <https://sv-comp.sosy-lab.org/>.

<sup>2</sup> <https://sv-comp.sosy-lab.org/2017/rules.php>.

should be a single preprocessed C file. Just few static verification tools support verification tasks that consist of several source files.

## 2.2 Checking Requirements

In this paper we focus on static verification of programs against non-functional requirements. These requirements include generic rules of safe programming suitable for any program. Violations of these rules cover such errors as buffer overflows and null pointer dereferences. Also, we consider as non-functional requirements specific rules of correct usage of the particular program API which violations are also quite widespread [9]. Below we refer to both generic and specific rules as *correctness rules*.

Static verification tools can check safety and liveness properties. To run a tool the SV-COMP community first proposes to describe a safety properties specification as a temporal logic formula. Currently, the formula can represent only a hard-coded set of safety properties corresponding mostly to generic rules of safe programming:

- error function unreachability,
- valid memory deallocations, pointer dereferences and memory tracking (to search such errors as buffer over-reads and over-writes, null pointer dereferences, uses after free and memory leaks),
- absence of integer overflows,
- program termination.

For verifying other correctness rules a user can use the following two options. The first one is weaving an additional code into a program either manually or automatically to express requirements using one of the supported safety properties. For instance, correct API usage rules can be formulated as unreachability of an error function. Exploiting means of particular static verification tools is the second option. For instance, CPAchecker allows to specify requirements to a program using automata that guide analysis [10]. However, this approach prevents using other static verification tools and it is out of the scope of this paper. Also in addition to the SV-COMP safety property specifications some tools support checking of other requirements such as absence of data races [11].

Static verification tools allow checking safety property specifications one by one and most of them terminate after finding a first violation. That is why it is better to check substantially different correctness rules independently. Nevertheless, it makes sense to express closely related requirements using the same safety property to avoid analysis of similar results, say, false alarms due to the same reasons, and to decrease computational resources consumption.

## 2.3 Analysis Accuracy

Static verification tools construct a program model in a sound way that keeps all errors present in a source code under verification. Also, they check specified

safety properties at all possible paths including poorly tested ones. Due to this proving model correctness means that there is no erroneous paths in the program.

However, to deal with real programs developed in the GNU C programming language static verification tools usually make some assumptions. Many tools can treat undefined functions as functions without side effects returning any value of corresponding return types. According to our knowledge, no static verification tool has support of the inline assembler. Some tools, e.g. implementing Bounded Model Checking, unroll loops to a given number of iterations. These assumptions can result in both missing bugs and false alarms, but hopefully users can change parameters and provide models to bypass these issues.

## 2.4 Performance and Scalability

Static verification is an extremely complicated problem and even the best static verification tools have poor scalability at verification of large programs. Usually, it is difficult to predict computational resources required for verification since an outcome depends on many factors such as a code complexity, a safety property being checked, verification algorithms, SMT and interpolation solvers. At the SV-COMP competition each verification task is executed under the following limits: 15 min of CPU time and 15 GB of RAM. Most successful tools can cope with programs of several dozens of KLOC in size within these limits but not always. Significant increasing of the code complexity almost always results in an enormous growth of required computational resources and inability to proceed with the same verification scope and precision.

Static verification tools often implement algorithms sequentially because of their nature, a few tools can use multi-core CPUs or distributed computing and there is no tools that employ GPUs [12]. Parallel solving of a verification task helps to yield verdicts faster but usually there is a considerable overhead to share complicated internal data structures. Thus, we do not consider this use case in the given paper. To substantially speed up solution of many independent verification tasks, static verification tools are executed in parallel at IaaS or PaaS clouds and clusters [12].

## 2.5 Environment Modeling and Checking Program Fragments

Libraries that used by a program, other programs, user inputs, etc. constitute an environment that can influence a program execution. To verify the program it is necessary to provide an environment model to a static verification tool which represents certain assumptions about the environment. In practice it should call program functions, initialize and modify a heap and exported variables as the environment does. Also, the environment model should contain models of undefined functions which the program calls.

Our experience shows that usually some results can be obtained even without the very accurate environment model. To get an acceptable false alarm rate and sufficient code coverage to avoid missing bugs it can be necessary to refine it.



To achieve really high-quality results it is crucial to provide the precise environment model taking into account specifics of the checked safety property and the program under verification.

To drastically reduce consumption of computational resources it is possible to verify program fragments of a moderate size separately. A program fragment can contain several source files of the program and libraries or just particular functions from them. However, it becomes even more important to provide the appropriate environment model at verification of the program fragment to avoid missing bugs and false alarms.

The SV-COMP community does not provide any tools or formats for facilitating specifying the environment model in a commonly accepted way. So, the most generic approach to provide the environment model is to weave it as an additional C code into the target program or the program fragment. Particular tools can provide additional means to describe an environment model as annotations, formulas or automata [10, 13]. But we do not consider such the use case in the given paper.

## 2.6 Tools Execution

Each verification task solution may require a considerable amount of CPU time, RAM and disk space. The BenchExec benchmarking tool comes to the aid as it enables fair computational resources distribution, isolates tool runs, performs preliminary results processing and simplifies integration of new static verification tools that follow the SV-COMP rules [14]. BenchExec measures and reports consumed CPU time, RAM and disk space. If tools exceed allowed limits, it terminates them. BenchExec is the convenient and reliable tool to solve verification tasks using a single machine.

BenchExec contains wrappers for all static verification tools that participate in the SV-COMP competition. Wrappers allow abstracting from the static verification tools interface when describing sets of verification tasks and when processing obtained results. However, a user still has to provide parameters to tune algorithms for each tool individually because default parameters defined at tool wrappers often do not suit for practical applications. For each successful static verification tool run a corresponding wrapper provides a verdict meaning a checked safety properties specification is satisfied or not. For each failed run the wrapper provides a short failure reason like “timeout” or “parsing failure” while details are kept within log files.

## 2.7 Formal Confirmation and Manual Analysis of Results

Static verification tools can provide proofs and counterexamples in a machine-readable format on each successful run. There is a common format of correctness and violation witnesses<sup>3</sup>. The proposed witness validation technique establishes confirmation of such witnesses detecting spurious ones [15, 16]. The technique is

<sup>3</sup> <https://github.com/sosy-lab/sv-witnesses>.

widely used in SV-COMP, so today all static verification tools participating in the competition provide their results similarly.

Witnesses express proofs and counterexamples using observer automata. A violation witness contains an observer automaton corresponding to a counterexample error path that leads from a program start to a found error. By design this automaton can miss some details of the error path and even some its parts. A witness validation tool considers the observer automaton in combination with a control-flow automaton extracted from the program to check feasibility of the error path.

Correctness witnesses have almost the same format but present results of proving, e.g. invariants for loops, for many paths of the program. However, correctness witnesses do not contain any complete proofs or guaranties for users. For instance, it is even impossible to understand which parts of the program were verified.

Although witnesses can be automatically validated, users should investigate them manually to comprehend proofs and reasons of bugs and false alarms. According to our knowledge, just CPAchecker represents witnesses in a more user-friendly way, but this does not help much for large programs because of this representation contains too many details. These details could be hidden using a domain knowledge, e.g. that some statements correspond to the environment model or some statements are not relevant for the checked safety properties specification.

In addition to witnesses some static verification tools can provide code coverage which lists analyzed lines and functions of the program. For its visualization one can use standard tools like LCOV<sup>4</sup>.

BenchExec can provide to a user statistics on verdicts provided by static verification tools and on computational resources consumed by them in the form of tables and plots. Such results visualization finely presents performance and correspondence of obtained verdicts to ideal ones (ideal verdicts for SV-COMP verification tasks are known in advance). This suits pretty well for comparison of static verification tools at the competition. However, users need additional means convenient at practical use for evaluation of results obtained for their programs when ideal verdicts are unknown.

### 3 Automating Static Verification of GNU C Programs

We do not suggest solutions that allow to completely automate the static verification workflow from scratch for all programs developed in the GNU C programming language. Our primary goal is to suggest solid foundations that are generic enough to not restrict application to some specific software. However, the framework will support out of the box those program subclasses for which we or others will develop and implement appropriate algorithms. For other software it will be worth doing that to reduce required manual efforts, to increase results quality and to decrease demands for computational resources.

<sup>4</sup> <http://ltp.sourceforge.net/coverage/lcov.php>.

Following subsections consider steps of the proposed method. For first two steps we implicitly assume that everything that is generated automatically can be incrementally refined manually if one will find this necessary.

### 3.1 Decomposition of Programs

Most of industrial programs are quite large, so it can be hard or impossible to statically verify them against any non-trivial correctness rule under reasonable computational resource limitations. To increase chances to get a meaningful outcome we suggest decomposing target programs and perhaps libraries invoked by them into program fragments which were already introduced in the previous section. The most challenging problem at programs decomposition is to determine particular files for program fragments. Program fragments may vary from a single C file to all files of a program or even several programs and libraries.

We suggest implementing the following options in the static verification framework:

- Provide to a user means to describe program fragments explicitly. This is the most time-consuming but flexible approach that can help to gain the best results. It is especially suitable at verifying some particular version and configuration of a medium-sized program.
- Develop a generic algorithm that will automatically split a program into weakly connected parts of a specified size. In contrast to the previous approach this one considerably reduces manual work at the programs decomposition step. But it might be necessary to spend much more time for analysis of worse results.
- Implement an algorithm for a particular project taking into account its structure. This approach requires an extra time for implementation for each new kind of programs but it contributes to both automatic programs decomposition and good enough results. This way suits for verification of large programs or many various configurations and versions of the same program.

For both manual and automatic approaches for program fragments generation it is useful to extract a build commands base that helps to understand how program source files are combined together to form the final object files and executables. For particular version and configuration of the program this base should include the following information on build commands in the strict order of their execution:

- For all build commands of interest:
  - corresponding build tool names,
  - absolute paths to directories where they are executed,
  - input and output file names.
- For compilation commands:
  - corresponding versions of source files referred both explicitly (C files) and implicitly (header files),
  - preprocessor options.

For extracting the build commands base we propose to intercept build commands during a program build. Indeed, GNU C programs often have complicated build processes. Thus, it can be necessary to describe semantics of additional build commands that should be intercepted.

The build commands base can be collected either outside the framework or from within it. The former is preferable when users want to perform builds as they usually do and to incorporate static verification with continuous integration systems. The latter is better when users need to verify some particular versions and configurations of their software.

### 3.2 Verification Tasks Generation

As it was stated in the previous section a verification task should contain besides a program fragment an additional code that corresponds to an environment model and, if necessary, that expresses checked requirements using one of the supported safety properties. Also, particular correctness rules can require setting specific parameters for a chosen static verification tool. We propose to generate these additional code and tool parameters on the basis of specifications developed manually using appropriate domain specific languages.

For each particular pair of a program fragment and a correctness rule a set of specifications can be unique, but some specifications can be the same for different pairs. To avoid repeats during development of specifications we suggest to use templates. Also, we propose to reuse a generated code if it is the same for different program fragments or correctness rules.

We already proposed a method for generating an environment model part that invokes program fragment interfaces for Linux kernel modules [17]. Corresponding specifications allow describing complicated interactions for event-driven programs in a quite compact way. Also, we have been starting developing a more generic approach on the base of this method. For developing the additional code, that expresses checked requirements using one of the supported safety properties, and models of interfaces invoked by the program fragment we suggest using an aspect-oriented extension for the C programming language [18]. Corresponding specifications and tools allow weaving program fragments, e.g. redirect function calls and macro substitutions from the original source code to model functions.

The final steps of the verification tasks generation is preprocessing, that is usually performed together with weaving, and merging of preprocessed files together. For the latter we suggest to use CIL that is a source-to-source transformation tool allowing merging files developed in the GNU C programming language [19]. Besides, CIL performs many optimizations simplifying following analysis.

After all each verification task is a single GNU C file prepared for an immediate run of a static verification tool and a safety properties specification which comply with the SV-COMP rules. In addition, a name, a version and parameters of the given tool and an amount of computational resources that can be used for solution are specified.

To incorporate a domain knowledge within verification tasks, e.g. to distinguish the additional code from the original one and to emphasize statements that are the most relevant to checked requirements, we suggest to use special comments. These comments can be provided directly within specifications. Besides, they can be generated automatically.

### 3.3 Verification Tasks Solution and Results Processing

Verification tasks generation can take a considerable amount of time. Hence, we suggest to start solution of verification tasks as soon as they appear if there are enough computational resources. For solving a few verification tasks we propose to use a single powerful enough machine. To considerably reduce a total time for verification of many program fragments or/and correctness rules it is necessary to solve corresponding verification tasks at an IaaS cloud or at a cluster.

Monitoring of available computational resources and their fair distribution between verification tasks are responsibilities of a scheduler. The scheduler should respect verification task priorities specified by users. Also, the scheduler should support canceling solution of verification tasks since sometimes users can decide that they do not need to continue verification anymore.

Some verification tasks can require considerably less computational resources than requested by a user. To avoid useless reservation we suggest performing speculative scheduling trying to run a static verification tool with lesser limitations first. It is worth accumulating statistics for verification tasks solution to base scheduling on that ground. For instance, some correctness rules can be much easier for checking on average than other ones.

In case of using a cloud or a cluster the scheduler should allow connecting and disconnecting worker nodes. If a worker node goes down, we suppose to automatically reschedule terminated verification tasks solution.

For isolating static verification tool runs and for measuring and limiting computational resources consumed by them at a single machine we propose to use already mentioned BenchExec [14]. After BenchExec finishes, the framework should process its output and results from a static verification tool. We suggest retrieving a verdict, a consumed computational resources report, a witness, log files and other information like code coverage and statistics if so.

As far as witnesses can omit some details, we suggest adding them by considering witnesses together with the program from the corresponding verification task. Besides, we propose to enrich witnesses with domain knowledge annotations on the base of special comments generated at the previous step.

Regarding code coverage users can be interested in total code coverage for all program fragments rather than code coverage for individual verification tasks. Thus, we suggest uniting it for various correctness rules.

### 3.4 User Interface

Below we present different use cases of the static verification framework and discuss relevant user interfaces.

**Verification Processes Setup.** In this use case we assume that the framework already supports everything required for a program under verification, in particular appropriate correctness rule specifications are available.

To proceed to verification users should choose correctness rules to be checked and provide program fragments either by describing them manually or by choosing and configuring an appropriate algorithm (a target program should be provided in form of its source code or build commands base). In addition, to get better results for particular programs we suggest to incrementally tune various parameters for programs decomposition, verification tasks generation, verification tasks solution and results processing.

For providing data and parameters and for starting subsequent automatic static verification we propose to use a multiuser graphical interface shareable via a network. Project-specific interfaces, which assumes various forms and helpers, are most likely the most convenient way for users, but usually it is hard to develop them. Therefore, we suggest to support a file-based configuration that is flexible enough to cope with various programs. In addition, users should be able to start up verification using command-line tools providing some data like a build commands base. This use case is quite natural when one incorporates static verification with continuous integration systems.

**Expert Results Analysis.** To simplify analysis of results by experts we suggest extending the interface for verification processes setup with the following:

- Provide information on running and completed verification processes. For each verification process experts should be able to analyze data and parameters with which it was set up. For running verification processes the interface should present their progress: the number of already solved and the total number of verification tasks, elapsed time and approximate left time.
- Visualize witnesses, code coverage and failure descriptions. The primary goal of this visualization is to hide from experts as much irrelevant details as possible according to the domain knowledge.
- Show various statistics over results that can help to understand a picture in general. For instance, it can be very useful to see how many warnings were yielded for a particular verification process, what warnings correspond to bugs and to false alarms, what are the most significant reasons of false alarms and so on.
- Allow to evaluate results by associating them with marks that should be applied automatically for similar results such as witnesses and failure descriptions. Experts should be able to supply each mark with a detailed description and tags. To further simplify analysis we suppose to keep all history of marks changes.
- Support views allowing arranging data in a more convenient way and to filter out irrelevant results. For instance, experts may want to see just violations of a specific correctness rule or marks modified after some date.

It is worth noting that since static verification can take considerable time it does have sense to represent results to experts as soon as they appear. In this case they are able to proceed to analysis of results faster, in particular it is possible to understand that something was done wrong without waiting for all results.

**Developing Correctness Rule Specifications and Extending the Framework.** If verification process setup itself does not help to improve results quality, e.g. to increase code coverage or to decrease a false alarm rate, we suggest to incrementally improve correctness rule specifications. In case when the static verification framework does not cope well with specific programs out of the box, one can develop and implement more appropriate algorithms to be incorporated into the framework.

Users can perform both these activities using their favorite editors or IDEs. Also, we encourage to supply them with special domain specific language editors and a SDK for developing framework extensions. We propose the framework GUI to support simple extensions like hot plugging of new static verification tools.

To understand consequences of improvements we suggest users should be able to compare results and associated expert marks for different verification processes.

## 4 Implementation

We partially implemented the proposed method for automated static verification of programs written in the GNU C programming language within the Klever framework<sup>5</sup>. Klever is an open source project. The primary programming language is Python 3.4. Users can install Klever on various Linux distributions. Also, we implemented scripts for automatic deployment within an OpenStack cloud.

We use the Django framework for developing *Klever Bridge* that provides the web GUI for verification processes setup and expert results analysis. As a database system *Klever Bridge* supports PostgreSQL and MariaDB. For deploying the GUI users can use either Apache2 with mod\_wsgi or NGINX with Gunicorn. *Klever Bridge* already supports multiple users with different roles, fair results representation and automated results assessment. For the latter one can use one of the several algorithms for comparing violation witnesses and regular expressions for matching failure descriptions. At the moment *Klever Bridge* does not support visualization of correctness witnesses.

Users should specify for each cloud or cluster worker node how many CPU cores, RAM and disk space the framework can use for solving verification tasks. Enough computational resources should be reserved for generation of program fragments and verification tasks, results processing as well as for an operating system and other running services and applications.

<sup>5</sup> <https://forge.ispras.ru/projects/klever>.

There are three schedulers currently implemented:

1. *Klever Native Scheduler* provides means to solve verification tasks using a single machine. For monitoring available computational resources and running services it uses Consul.
2. *Klever Docker Scheduler* can solve verification tasks within a cluster or a cloud by leveraging an infrastructure for Docker containers.
3. *Klever VerifierCloud Scheduler* submits verification tasks to VerifierCloud<sup>6</sup>.

At the moment verification tasks can be solved with help of CPAchecker and Ultimate Automizer [20]. To integrate new static verification tools within Klever users need to do the following:

1. Describe specific options suitable for checking corresponding correctness rules.
2. Provide static verification tool binaries in case of using *Klever Native Scheduler*.
3. For *Klever Container Scheduler* install tools within Docker images and push these images to a Docker registry.

Currently, fully automatic programs decomposition and generation of verification tasks are available only for Linux kernel loadable modules as a proof of concept. The framework extracts the build command base itself allowing users to guide the build process via parameters. Users can choose one of the several algorithms for program fragments generation. The main one is to verify each Linux kernel loadable module separately. Another one allows specifying files and modules to unite manually. It is also possible to generate program fragments for groups of modules fully automatically on the base of dependencies between modules using a greedy algorithm.

The framework generates verification tasks in parallel to speed up the entire verification process when many computational resources are available. As a source code querier and a weaver we use CIF [18]. It is a source-to-source weaver that is based on GCC, and, thus, it can handle GNU C programs. CIF allows to perform a variety of structural source code queries and support weaving of macro substitutions and function calls.

The environment model is described as a parallel composition [17]. It is translated into a C code using an additional information extracted by querying a source code of a target program. Utilization of different translators allows generating either a parallel or sequentialized environment model. It also allows making heuristic simplifications of the model, e.g. to reduce interleaving. For Linux kernel loadable modules we have implemented environment model specifications to support interrupts, timers and interfaces of kernel subsystems including USB, PCI, SCSI, SERIAL, NET, file systems, etc.

We allow users to check a variety of correctness rules ranging from generic memory safety to correct usage of the most popular Linux kernel API. Also, one can check new requirements by developing additional specifications.

---

<sup>6</sup> <https://vcloud.sosy-lab.org/cpachecker/webclient/help/>.



## 5 Related Work

According to our knowledge, no static verification framework automates preparation of an arbitrary GNU C program before static verification, runs static verification tools, processes results and provides means for their further analysis and improvement. There are a few projects that focus on automated static verification of specific software.

SDV is the best-known application of static verification in practice [21]. It aims at checking correct usage of the kernel API in Windows drivers using SLAM, YOGI and Q [1, 22]. There are also LDV Tools [23], DDVerify [24], Avinux [25] frameworks intended for static verification of Linux drivers. As a result, hundreds of bugs have been found and acknowledged by driver developers already.

CBMC [2] has been applied for verification of TinyOS [26] and embedded software [27]. Authors deliver successful case studies as a proof of concept.

DC2 is a framework that aims at static verification of industrial software [13]. To bound a verification scope it generates contracts relevant for safety properties like memory leaks and array-bound overflows. If necessary, users can improve these contracts manually. Then DC2 runs the Varvel model checker. However, it is the in-house NEC research project, so it is not possible to estimate its applicability to software developed in the GNU C programming language in more details.

There is an IDE for development of embedded software *mbeddr* that allows to automatically run CBMC to check programs under development against a predefined set of safety properties [28]. The IDE also provides developers with nicely arranged results. However, *mbeddr* is not intended for automated static verification of programs developed outside of it.

## 6 Conclusion

We presented the method that addresses problems of automated application of static verification tools for checking programs developed in the GNU C programming language. This method has been partially implemented within the Klever framework that already demonstrated its applicability to large industrial software projects like Linux kernel loadable modules. To complete the research, we are going to provide comprehensive evaluation verifying various programs.

We based Klever on solutions accepted by the SV-COMP community. Moreover, we keep in touch with it to cooperate and to solve the most vital problems together discussing the interface, providing a feedback and contributing generated verification tasks to the competition benchmark suit.

## References

1. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, pp. 35–42. FMCAD Inc, Austin (2010)
2. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-24730-2\\_15](https://doi.org/10.1007/978-3-540-24730-2_15)
3. Beyer, D., Keremoglu, M.E.: CPACHECKER: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-22110-1\\_16](https://doi.org/10.1007/978-3-642-22110-1_16)
4. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. *Adv. Comput.* **58**, 117–148 (2003)
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* **50**(5), 752–794 (2003)
6. Beyer, D.: Competition on software verification. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-28756-5\\_38](https://doi.org/10.1007/978-3-642-28756-5_38)
7. Beyer, D.: Software verification with validation of results. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). [https://doi.org/10.1007/978-3-662-54580-5\\_20](https://doi.org/10.1007/978-3-662-54580-5_20)
8. Haran, A., Carter, M., Emmi, M., Lal, A., Qadeer, S., Rakamarić, Z.: SMACK+Corral: a modular verifier. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 451–454. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_42](https://doi.org/10.1007/978-3-662-46681-0_42)
9. Mutilin, V.S., Novikov, E.M., Khoroshilov, A.V.: Analysis of typical faults in Linux operating system drivers. *Proc. ISP RAS* **22**, 349–374 (2012)
10. Apel, S., Beyer, D., Mordan, V., Mutilin, V., Stahlbauer, A.: On-the-fly decomposition of specifications in software model checking. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 349–361. ACM, New York (2016)
11. Andrianov, P.S., Mutilin, V.S., Khoroshilov, A.V.: Predicate abstraction based configurable method for data race detection in Linux kernel. In: Itsykson, V., Scedrov, A., Zakharov, V. (eds.) TMPA 2017. CCIS, vol. 779, pp. 11–23. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-71734-0\\_2](https://doi.org/10.1007/978-3-319-71734-0_2)
12. Zakharov, I.S.: A survey of high-performance computing for software verification. In: Itsykson, V., Scedrov, A., Zakharov, V. (eds.) TMPA 2017. CCIS, vol. 779, pp. 196–208. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-71734-0\\_17](https://doi.org/10.1007/978-3-319-71734-0_17)
13. Ivančić, F., Balakrishnan, G., Gupta, A., Sankaranarayanan, S., Maeda, N., Imoto, T., Pothengil, R., Hussain, M.: Scalable and scope-bounded software verification in varvel. *Autom. Softw. Eng.* **22**(4), 517–559 (2015)
14. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Fischer, B., Geldenhuys, J. (eds.) SPIN 2015. LNCS, vol. 9232, pp. 160–178. Springer, Cham (2015). [https://doi.org/10.1007/978-3-319-23404-5\\_12](https://doi.org/10.1007/978-3-319-23404-5_12)
15. Beyer, D., Dangel, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp. 721–733. ACM, New York (2015)

16. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 326–337. ACM, New York (2016)
17. Khoroshilov, A., Mutilin, V., Novikov, E., Zakharov, I.: Modeling environment for static verification of Linux Kernel modules. In: Voronkov, A., Virbitskaite, I. (eds.) PSI 2014. LNCS, vol. 8974, pp. 400–414. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46823-4\\_32](https://doi.org/10.1007/978-3-662-46823-4_32)
18. Novikov, E.M.: An approach to implementation of aspect-oriented programming for C. *Program. Comput. Softw.* **39**(4), 194–206 (2013)
19. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45937-5\\_16](https://doi.org/10.1007/3-540-45937-5_16)
20. Heizmann, M., Dietsch, D., Leike, J., Musa, B., Podelski, A.: ULTIMATE AUTOMIZER with array interpolation. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 455–457. Springer, Heidelberg (2015). [https://doi.org/10.1007/978-3-662-46681-0\\_43](https://doi.org/10.1007/978-3-662-46681-0_43)
21. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. *Commun. ACM* **54**(7), 68–76 (2011)
22. Lal, A., Qadeer, S.: Powering the static driver verifier using corral. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 202–212. ACM, New York (2014)
23. Zakharov, I.S., Mandrykin, M.U., Mutilin, V.S., Novikov, E.M., Petrenko, A.K., Khoroshilov, A.V.: Configurable toolset for static verification of operating systems kernel modules. *Program. Comput. Softw.* **41**(1), 49–64 (2015)
24. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent Linux device drivers. In: Proceedings of the 22nd International Conference on Automated Software Engineering, pp. 501–504. ACM, New York (2007)
25. Post, H., Kuchlin, W.: Integrated static analysis for Linux device driver verification. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 518–537. Springer, Heidelberg (2007). [https://doi.org/10.1007/978-3-540-73210-5\\_27](https://doi.org/10.1007/978-3-540-73210-5_27)
26. Bucur, D., Kwiatkowska, M.Z.: Software verification for TinyOS. In: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, pp. 400–401. ACM, New York (2010)
27. Schlich, B., Kowalewski, S.: Model checking C source code for embedded systems. *Int. J. Softw. Tools Technol. Transf.* **11**(3), 187–202 (2009)
28. Cârlan, C., Ratiu, D., Schätz, B.: On using results of code-level bounded model checking in assurance cases. In: Skavhaug, A., Guiochet, J., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2016. LNCS, vol. 9923, pp. 30–42. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-45480-1\\_3](https://doi.org/10.1007/978-3-319-45480-1_3)

# Domain Specific Semantic Validation of Schema.org Annotations

Umutcan Şimşek<sup>(✉)</sup>, Elias Kärle, Omar Holzkecht, and Dieter Fensel

STI Innsbruck, University of Innsbruck, Technikerstrasse 21a, 6020 Innsbruck, Austria  
{[umutcan.simsek](mailto:umutcan.simsek@sti2.at),[elias.kaerle](mailto:elias.kaerle@sti2.at),[omar.holzkecht](mailto:omar.holzkecht@sti2.at),[dieter.fensel](mailto:dieter.fensel@sti2.at)}@sti2.at  
<http://www.sti2.at>

**Abstract.** Since its unveiling in 2011, schema.org has become the de facto standard for publishing semantically described structured data on the web, typically in the form of web page annotations. The increasing adoption of schema.org facilitates the growth of the web of data, as well as the development of automated agents that operate on this data. Schema.org is a large heterogeneous vocabulary that covers many domains. This is obviously not a bug, but a feature, since schema.org aims to describe almost everything on the web, and the web is huge. However, the heterogeneity of schema.org may cause a side effect, which is the challenge of picking the right classes and properties for an annotation in a certain domain, as well as keeping the annotation semantically consistent. In this work, we introduce our rule based approach and an implementation of it for validating schema.org annotations from two aspects: (a) the completeness of the annotations in terms of a specified domain, (b) the semantic consistency of the values based on pre-defined rules. We demonstrate our approach in the tourism domain.

**Keywords:** Rule-based systems · Semantic validation · Schema.org

## 1 Introduction

To publish structured data on the web there are a lot of collections of vocabularies and ontologies that all serve a different or overlapping purpose and appear, grow and vanish in an unpredictable manner. However, there is one initiative to provide structured data on the web which stands out by means of community adoption and distribution and became a de facto standard, which is schema.org<sup>1</sup>. Schema.org was developed in 2011 by Google, Microsoft, Yahoo! and Yandex and has been supported since by a broad community and found application on millions of websites [2]. Schema.org can be included into the website's source code with common technologies like Microdata, RDFa or JSON-LD. The vocabulary covers local businesses, products, events, recipes, people and much more and is adapted and supported by the big search engine providers. This naturally makes

---

<sup>1</sup> <https://schema.org>.

the vocabulary quite heterogeneous. The vocabulary is also semantically imperfect [9]. For instance classes may inherit properties improperly (e.g. a waterfall can have a telephone number) and not formally strict, but this is rather a design decision to facilitate rapid and decentralized evolution of the vocabulary. The side effect of this feature is that picking the right classes and properties for a domain can be quite challenging and low quality annotations in terms of conforming to the rules of a field (e.g. tourism) may occur.

The World Wide Web was originally designed as an internet-based hypertext system. It contains blocks of information, the websites, which are connected via hyperlinks to other blocks of information. Due to that simple design and the open-to-all approach it rapidly evolved to be the biggest information network that ever existed. The headless web<sup>2</sup> is a layer which grows on top of the Web we know. Within this layer goods are not sold by individual producers or small retail websites, but by a few large retail platforms like Alibaba or Amazon. Rooms are not sold by hotels or destination marketing organizations (DMOs) but by a hand full of huge online travel agencies (OTAs) like booking.com or Expedia. In a not too distant future information will no longer be found on individual websites, but gathered by the search engines and presented to the searching user directly on the search engine website. So the web is, in the true sense of the word, losing its head: its graphical representation. The data will be extracted from websites and presented to the user not only by the search engines but also by personal assistant software like Cortana, Siri, or Google Now. With this new layer we can observe a trend towards going-out-of-use of graphical representation and the rising necessity of structured, high quality, data. The data for services like Cortana or Siri is going to be collected and gathered by crawlers and only structured, machine read- and understandable data will be part of the game at that point. In the headless web there will be no room for unstructured but beautifully designed content. The challenge for small and medium enterprises (SME) is to bring their data into this new layer by precise, correct and complete semantic annotations on their websites. Schema.org is the vocabulary of choice to do that and hence SMEs need a way to produce schema.org annotations in a correct way and a tool to validate those annotations.

This paper describes such a method to define domain specific subsets of the schema.org vocabulary with enriched semantics and also introduces the tool we provide in order to validate the semantics of domain specific structured data annotated with schema.org on websites. Depending on the domain, a subset of schema.org classes and properties will be selected and a set of rules will be defined by a domain expert - which is the foundation of the validation process. From there on users can validate their own annotations and websites based on the domain specific subset and the validation rules defined by the domain expert.

---

<sup>2</sup> <https://paul.kinlan.me/the-headless-web/>.

The remainder of this paper is organized as follows: Sect. 2 compares the described approach with related work. Section 3 describes our method which includes a domain definition and validation approach and a tool that implements it. Section 4 shows the approach in action and Sect. 5 gives an outlook to future work and concludes the paper.

## 2 Related Work and Motivation

While the adoption of schema.org has been increasing [8], the conformance of the schema.org annotations to the vocabulary specification is still questionable. A large scale study on the usage of schema.org in the tourism domain [3] shows that the schema.org vocabulary is mostly used incorrectly or missing fundamental properties (e.g. many hotels do not have address information in their annotations). The issue of completeness for the schema.org annotations occurs due to the size of the vocabulary and the lack of guidance for adopters to decide which classes and properties to use. In addition to this issue, there is also the semantic consistency issue (e.g. consistency between the country and the country code of a phone number) for annotations that is not possible to capture with the prominent validation tools like the Google Structured Data Testing Tool<sup>3</sup>.

Given the developments about the new layer on top of the web, providing well formed and semantically consistent structured data on the web is more important than ever. Therefore, we propose an approach, that allows us to obtain a specific subset of the schema.org vocabulary containing important classes and properties for a domain and to validate the annotations based on pre-defined rules to ensure the completeness and the semantic correctness of the data.

The related work to our approach comes mostly from the RDF validation domain. An approach described in [1] applies SPIN Rules for domain independent detection of certain data quality problems namely, inconsistency (i.e. inconsistent representation of the data, functional dependency and referential integrity), comprehensibility (i.e. ambiguity of the data), heterogeneity and redundancy. An approach [11] presented in the RDF Validation Workshop [7] proposes a simple mechanism for declaring the properties to be used for a class and a SPARQL based extension for defining more complex constraints. Parallel to the RDF Validation Workshop results, there have been an increased development of new RDF validation methods. Shape Expressions (ShEx) [10] is a domain specific language for validating and transforming RDF Data. Similar to ShEx, RDF Data Shapes Working Group has been developing the Shapes Constraint Language (SHACL) [6] for describing and validating RDF graphs. SHACL allows us to define constraints targeting specific nodes in a data graph based on their type, identifier, or a filtering SPARQL query. It is currently investigated that at what level ShEx can be represented in SHACL, based on the identified similarities and differences<sup>4</sup>. The rule-based validation of RDF data is an emerging field, mostly focused around the re-use of prominent standards like SPARQL.

<sup>3</sup> <https://search.google.com/structured-data/testing-tool>.

<sup>4</sup> <http://shex.io/primer/#rel-to-shacl>.

All of the aforementioned validation approaches are somewhat compatible with SPARQL. Our approach shows similarities with aforementioned approaches in terms of using rules for checking consistency of the data and defining constraints over classes. The works in [6, 10] allow us to define “shapes” that constraint types and instances in terms of subset of properties and expected types for those properties as well as nested shapes.

We introduce the notion of “domain” and a simple specification of it for schema.org, which adopts a similar nested definition of constraints that restricts classes and properties in relation to other classes of which they are expected types. The concept of selecting a subset of schema.org appears in [5], but to the best of our knowledge, the domain selection of the editor described there is limited to the selection of classes. We propose a different domain specification approach including selecting a subset of properties and restricting the range of those properties to a subset of subclasses of the range defined by schema.org. The importance of this restriction is described in Sect. 3.1 in more detail. Additionally, our validator brings domain definition and semantic consistency rules together in one holistic tool.

In order to show a concrete example of our motivation for domain specific validation, we can consider annotation of an event. The *Event* class of schema.org vocabulary contains 38 properties including the ones inherited from the *Thing* class. Even though this number seems not too high, the properties whose range is a complex type makes the annotation size unmanageable. Let us take only one property of the Event class into account: organizer. This property can have values in the *Organization* class. If a user starts to annotate an event and its organizer, she will soon realize that the *Organization* class itself offers 50 properties. The amount of properties and classes the user needs to deal with explodes as we continue. When we define a domain, we can select a subset of properties of the *Organization* class as the value of the organizer property, for instance, to only name and url. This restriction of classes when they are the value of a certain property will give a clear idea to the user who creates schema.org annotations.

### 3 Method

In this section, we explain our approach in detail and demonstrate the web based tool<sup>5</sup> that implements it.

Our approach consists of two main parts. First, the definition of a domain by selecting a subset of classes and properties (Sect. 3.1) as well as a set of semantic validation rules (Sect. 3.2). Second, the creation and validation of a schema.org annotation in terms of its completeness regarding the defined domain and semantic consistency based on the validation rules (Sect. 3.3).

---

<sup>5</sup> <http://sdo-validator.sti2.at>.

### 3.1 Domain Definition

A domain expert, who has an extensive knowledge in a certain field (e.g. tourism), defines a domain by selecting a subset of the schema.org vocabulary, the classes and properties, which is relevant to a certain domain. Moreover, it can be specified whether a property is required for a concept or allowed to have multiple values. The domain definition process consists of the following steps: First, the domain expert selects a subset of schema.org classes. Second, she specifies the allowed properties for the selected classes, as well as whether they are optional or allowed to have multiple values. In step three, for every property added into the domain, she selects the expected types of the property. She continues the domain specification by recursively following the aforementioned steps for complex types (e.g. If the address property of a Hotel is included to the domain and its expected value type is PostalAddress, the same process should be applied also for the PostalAddress class) until the domain is complete.

In order to facilitate the domain definition, we developed the Domain Definition Interface (Fig. 1) as a part of our tool. The aforementioned steps can be applied via the interface to create a domain. After the domain expert completes the domain, the tool generates a JSON file which contains the domain specification.

A domain specification consists of classes, that contains properties whose expected values can be in unrestricted classes (i.e. schema.org/Class) and restricted classes (e.g. a class with only a subset of its properties). Every restricted class is based on a schema.org/Class. The expected types of a property can also be restricted to a certain subset of their subclasses. Being able to restrict expected types to a subset of subclasses would be especially useful for properties like *schema.org/potentialAction*, since its range is the Action class which is the most generic action. However, for a specific domain, a certain class may be required to have more specific actions as its potential action (e.g. The *schema.org/potentialAction* of the *schema.org/HotelRoom* class may be restricted to *schema.org/ReserveAction*). A concrete example of a domain can be found in Sect. 4.

### 3.2 Rule Definition

Rules are created by domain experts. In order to define a rule, the domain expert first has to select a predefined domain or create a new one. Then she can create the set of rules applying to the defined domain. A semantic validation rule is a condition-action rule where an action is triggered when a condition is satisfied. Since these rules are used for validation, the condition part of a rule must state the condition that violates the domain requirement and the action part should contain the action that will be taken when the condition is satisfied (i.e. domain requirement is violated). Domain experts may use the concepts and properties that are allowed in the domain definition (Sect. 3.1), Boolean and arithmetic operations as well as some predefined utility functions. In some cases, rules might require more complex processing of the data. To achieve this, domain



Process properties of Class

Actual Class: LodgingBusiness

Description: Thing / Organization / LocalBusiness / LodgingBusiness  
Thing / Place / LocalBusiness / LodgingBusiness

A lodging business, such as a motel, hotel, or inn.

Subclass selection:  Allow Selection of Subclasses

LodgingBusiness
 

- BedAndBreakfast
- Motel
- Hotel
- Hostel
- Resort
- Campground

Property Name	Allowed value types	Visibility	Allow Multiple Values
additionalProperty ⓘ	<input checked="" type="checkbox"/> PropertyValue <span style="float: right; font-size: small;">EDIT</span>	Required <span style="float: right;">👁️ 👁️ 👁️</span>	<input checked="" type="checkbox"/> <input checked="" type="checkbox"/>
additionalType ⓘ	<input checked="" type="checkbox"/> URL	Required	<input type="checkbox"/>
address ⓘ	<input checked="" type="checkbox"/> PostalAddress <span style="float: right; font-size: small;">EDIT</span>	Required	<input type="checkbox"/>

**Fig. 1.** A screenshot from the domain definition interface. Here, a domain expert can select a subset of properties and define restrictions on them and their expected types

experts can define their own utility function (e.g. a function that looks up for the international country calling code for a given country). We introduce two different type of condition-action rules: local consistency and global consistency rules. Local consistency rules compare the value of a property with a literal value (e.g. The floor size of a room must be greater than zero). An example of the local consistency rule is shown in Listing 1.1.

```

Condition:
HotelRoom.floorSize.QuantitativeValue.value <= 0
Action:
show("Floor size of a hotel room must be greater
    than zero.", Severity:Error)

```

**Listing 1.1.** “An informal representation of a local consistency validation rule”

A global consistency rule is involved with multiple properties. These properties can originate from the same class or from different classes. The following example explains the elements of a global consistency rule: A domain expert may want to create a validation rule that checks if the international country calling code of a telephone number is consistent with the country in the postal address. Such an informal validation rule may look like the Listing 1.2.

```
Condition:
extractCountryCode(Place.telephone) !=
    getCountryCodeByCountry
        (Place.address.PostalAddress.addressCountry)
Action:
show("The international country code of the phone
    number of the place is not consistent with the
    country of the address.", Severity:Error)
```

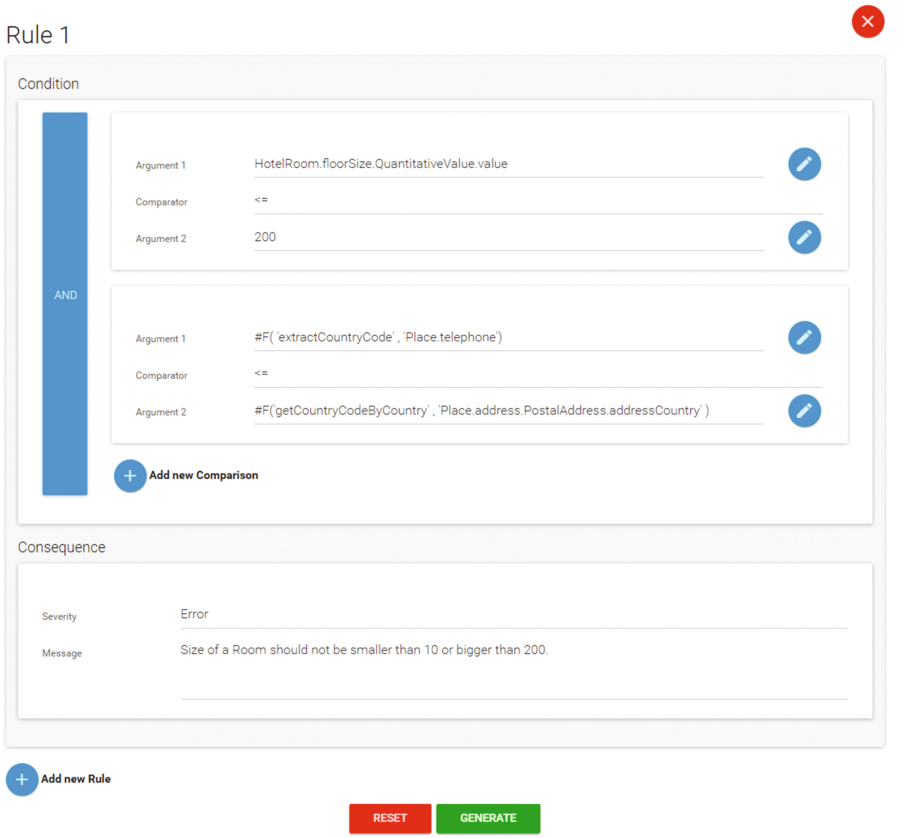
**Listing 1.2.** “An informal representation of a global consistency validation rule”

In the condition part, a utility function called “extractCountryCode” takes the value of the telephone property of a *Place* instance as parameter and returns the international country calling code. Another utility function called “getCountryCodeByCountry” takes the value of the *addressCountry* property of a *PostalAddress* instance of the same *Place* instance and returns the international country calling code for the specified country. If the comparison shows that two values are not equal, the Action part is triggered. The predefined utility function “show” displays the reason and the severity of the violation. Rules not only define what is allowed or what is not, but also gives meaningful correction suggestions like “The phone number you specified does not match the mentioned country. Is that really correct?”. These suggestions have to be defined in the rules as well.

Figure 2 shows the first prototype of the rule designer, which is a form based component of our tool to enable domain experts to create semantic validation rules. Via this interface, the domain expert can create rule conditions that represent semantic inconsistencies and suitable error messages to show, in case the violation conditions are satisfied.

### 3.3 Annotation and Validation

In order to guide a user who wants to create an annotation in a certain domain, we generate an annotation editor based on a domain specification and ensure the completeness of the annotation. An annotation is valid in terms of completeness if it contains all required properties, none of the unspecified properties, and correct expected types for the properties defined in the domain and used in the annotation.



**Fig. 2.** Prototypical interface of the Rule Designer

The annotation then can be validated for semantic consistency. The validation process iterates over all the rules defined and saves the result of the validation against each rule in a list to be presented to the user. Similar to the definition of the rules, we distinguish between local and global consistency rules. Local consistency rules consider the value of only one property, global consistency rules consider the values of several properties, check complex relations between various properties, and can go over several rules.

Figure 3 depicts the validation interface of our tool, which is used by the user for validation of an annotation. This interface can validate an annotation for both completeness and semantic consistency.<sup>6</sup> The validator first ensures the

<sup>6</sup> For the annotations that are created via the editor based on the domain specification, only the semantic consistency validation applies.

### Specific Domain Definition Validation

Paste your Specific Domain Definition here and then press the "Validate" button.

1. Specific Domain Definition

```
{
  "@type": "schema:DataType",
  "@id": "schema:Text",
  "schema:name": "Text"
},
{
  "@type": "schema:DataType",
  "@id": "schema:Boolean",
  "schema:name": "Boolean"
}
}
```

2. Rules Definition

```

{
  "message": "Size of a Room should not be smaller than 10 or bigger than 200."
},
{
  "condition": "[#F( extractCountryCode , 'Place telephone');!#F(getCountryCodeByCountry , 'Place address.PostalAddress addressCountry')]",
  "consequence": {
    "severity": "Error",
    "message": "Telephone number does not match with the selected country."
  }
}
}
```

**Not valid!**  
Telephone number does not match with the selected country

3. Specific Domain Annotation

```

{
  "name": "geo:name"
},
{
  "image": "https://en.wikipedia.org/wiki/Main_Page",
  "legalName": "Hotel EU",
  "logo": "https://en.wikipedia.org/wiki/Main_Page",
  "name": "Hotel eu",
  "petsAllowed": false,
  "priceRange": "30 - 500",
  "smokingAllowed": true,
  "telephone": "80256409",
  "url": "https://en.wikipedia.org/wiki/Main_Page"
}
}
```

RESET VALIDATE

**Valid!**  
Your input is a valid Schema.org Annotation according to the given Specific Domain Definition.

**Fig. 3.** Validation interface

syntactic correctness of the entries. Then it validates the completeness of the annotation. If the annotation conforms the domain specification, the validator iterates over the rules defined in the rule set and warns the user if there is any semantic inconsistency within the annotation.

### 4 Use Case: Annotation of a Lodging Business

In order to demonstrate our approach and implementation, we created the domain represented in Fig. 4 and semantic validation rule in Listing 1.2 via the domain definition interface and rule designer depicted in Fig. 2.

In our scenario, a user wants to validate the annotation for Moosleite in Mayrhofen (Listing 1.3) against the domain specification and semantic validation rule. When the user enters the domain specification and rule set to the validator and then validates the annotation, she receives a completeness error. This is because the domain requires the *currenciesAccepted* property but the annotation does not have it.

After the addition of the missing required property to the annotation, the rule-based validation takes place. The semantic validation rule validates whether the country code of the phone number is consistent with the country of the address. Since this is not the case, the user receives the “*The international country code of the phone number of the place is not consistent with the country of the address.*” error message defined in the action part of the rule in Listing 1.2. When the country code of the telephone number is also corrected, the user receives the confirmation that the annotation is valid.

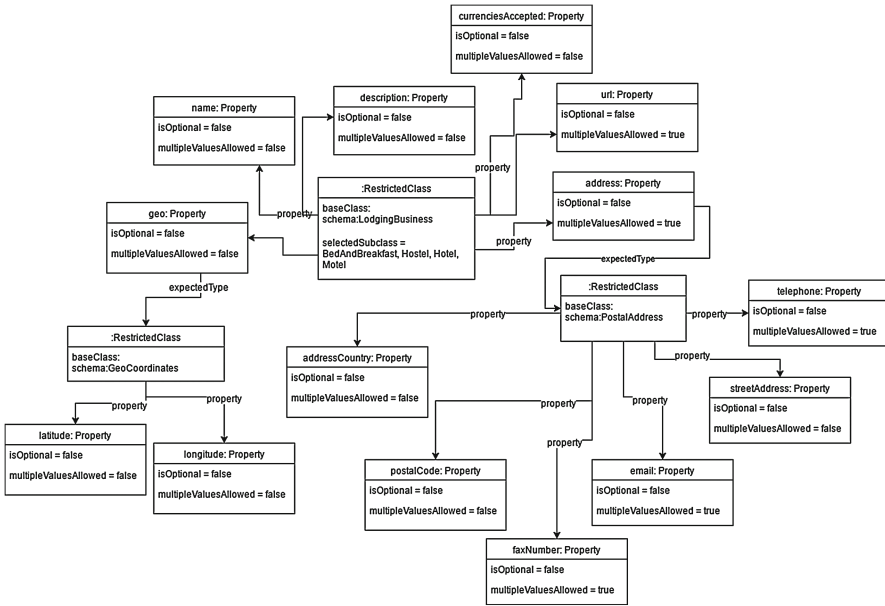


Fig. 4. A domain definition for lodging businesses

```
{
  "@context": "http://schema.org",
  "@type": "LodgingBusiness",
  "url": [
    "http://www.tiscover.com/moosleite",
    "http://maps.mayrhofen.at/?foreignResource=E33CFC29-050E-43D7-9BB3-EA937D33FCA4"
  ],
  "address": {
    "@type": "PostalAddress",
    "postalCode": "6290",
    "streetAddress": "Neu-Burgstall 318",
    "addressCountry": "AT",
    "telephone": "+42 5285 62894",
    "email": "eberl.friedl@tirol.com",
    "faxNumber": "0043 5285 62064",
    "url": "http://www.tiscover.com/moosleite"
  },
  "name": "Moosleite",
  "description": "Our house is situated approx. 1.5km from
    Mayrhofen, at the edge of the forest and enjoying
    wonderful panoramic views.",
  "geo": {
    "@type": "GeoCoordinates",
    "latitude": "47.1862746335978",
    "longitude": "11.8581855297089"
  }
}
```

**Listing 1.3.** An example annotation of Moosleite Hotel Mayrhofen. The country code of the phone number does not match the country of the address and the `currenciesAccepted` property is missing.

## 5 Conclusion and Future Work

The web we know is changing and the only way to remain visible on the new layer of the web is providing semantically described structured data. Schema.org is helping us to achieve this goal since 2011 as the de facto standard for describing things on the web.

We acknowledge that schema.org adopts “some data better than no data” motto and its data model is imperfect by its nature<sup>7</sup>. However, it is still important to publish high quality structured data that conforms to the schema.org vocabulary. We aim to help users for achieving this goal with our domain specific validation approach. In this paper, we introduced a domain specific approach to validate schema.org annotations. Our approach allows domain experts to specify a domain based on a subset of schema.org vocabulary as well as validation rules for semantic consistency. We showed the web based implementation of our approach alongside a use case in the tourism area.

<sup>7</sup> <http://schema.org/docs/datamodel.html>.

For the future work we will follow the works of different groups, especially the RDF Data Shapes Working Group, to find out possible alignments between our approaches. For instance, development in the SHACL shows promising results and can be utilized for the later implementation of our approach.

Moreover, we are in the processes of advancing the tool that implements our approach while including the development of more sophisticated rule designer and validator. We will test our tool in a larger scale in tourism domain within the next months.

Our approach currently does not consider multi-typed entities, which are encouraged by the schema.org initiative. For instance, the schema.org hotel extension [4] suggests that a lodging business should define their rooms as both *schema.org/Room* and *schema.org/Product* in order to conform schema.org specifications. We will investigate how we can adopt the multi-typed entity notion in the future work.

## References

1. Fürber, C., Hepp, M.: Using SPARQL and SPIN for data quality management on the semantic web. In: Abramowicz, W., Tolksdorf, R. (eds.) BIS 2010. LNBI, vol. 47, pp. 35–46. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-12814-1\\_4](https://doi.org/10.1007/978-3-642-12814-1_4)
2. Guha, R.V., Brickley, D., Macbeth, S.: Schema.org: evolution of structured data on the web. *Commun. ACM* **59**(2), 44–51 (2016). <http://doi.acm.org/10.1145/2844544>
3. Kärle, E., Fensel, A., Toma, I., Fensel, D.: Why are there more hotels in Tyrol than in Austria? Analyzing Schema.org usage in the hotel domain. In: Inversini, A., Schegg, R. (eds.) *Information and Communication Technologies in Tourism 2016*, pp. 99–112. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-28231-2\\_8](https://doi.org/10.1007/978-3-319-28231-2_8)
4. Kärle, E., Şimşek, U., Akbar, Z., Hepp, M., Fensel, D.: Extending the Schema.org vocabulary for more expressive accommodation annotations. In: Schegg, R., Stangl, B. (eds.) *Information and Communication Technologies in Tourism 2017*, pp. 31–41. Springer, Cham (2017). [https://doi.org/10.1007/978-3-319-51168-9\\_3](https://doi.org/10.1007/978-3-319-51168-9_3)
5. Khalili, A., Auer, S.: WYSIWYM authoring of structured content based on Schema.org. In: Lin, X., Manolopoulos, Y., Srivastava, D., Huang, G. (eds.) *WISE 2013*. LNCS, vol. 8181, pp. 425–438. Springer, Heidelberg (2013). [https://doi.org/10.1007/978-3-642-41154-0\\_32](https://doi.org/10.1007/978-3-642-41154-0_32)
6. Knublauch, H., Kontokostas, D.: Shapes constraint language (2016). <https://w3c.github.io/data-shapes/shacl/>
7. Le Hors, A., Solbrig, H., Prudhommeaux, E.: RDF validation workshop report, practical assurances for quality RDF data. Technical rep., Cambridge, MA, USA (2013). <https://www.w3.org/2012/12/rdf-val/report>
8. Meusel, R., Bizer, C., Paulheim, H.: A web-scale study of the adoption and evolution of the Schema.org vocabulary over time. In: *Proceedings of the 5th International Conference on Web Intelligence, Mining and Semantics, WIMS 2015*, pp. 15:1–15:11. ACM, New York (2015). <http://doi.acm.org/10.1145/2797115.2797124>
9. Patel-Schneider, P.F.: Analyzing Schema.org. In: Mika, P., et al. (eds.) *ISWC 2014*. LNCS, vol. 8796, pp. 261–276. Springer, Cham (2014). [https://doi.org/10.1007/978-3-319-11964-9\\_17](https://doi.org/10.1007/978-3-319-11964-9_17)

10. Prud'hommeaux, E., Labra Gayo, J.E., Solbrig, H.: Shape expressions: an RDF validation and transformation language. In: Proceedings of the 10th International Conference on Semantic Systems - SEM 2014, pp. 32–40 (2014)
11. Simister, S., Brickley, D.: Simple application-specific constraints for RDF models. In: RDF Validation Workshop. Practical Assurances for Quality of RDF Data, Cambridge, MA, Boston, pp. 1–5 (2013). <https://www.w3.org/2001/sw/wiki/images/0/00/SimpleApplication-SpecificConstraintsforRDFModels.pdf>



## Author Index

- Artemev, Vasiliï 1  
Askarpour, Mehrnoosh 12  
Avetisyan, Arutyun 28
- Baar, Thomas 394  
Belevantsev, Andrey 28  
Brass, Stefan 43  
Buchatskiy, Ruben 375
- Carvallo, Pamela 59  
Cavalli, Ana R. 59  
Chernenkiy, Valeriy M. 72  
Chernishev, George 88
- Dovgalyuk, Pavel 132  
Dragoni, Nicola 95
- Ermakov, Mikhail 105
- Fedorenko, Yuriy S. 72  
Fensel, Dieter 417  
Ferrarotti, Flavio 117  
Fursova, Natalia 132
- Galaktionov, Viacheslav 88  
Gapanyuk, Yuriy E. 72  
Garanina, Natalia 147  
Gorlatch, Sergei 179  
Grigorev, Valentin 88  
Grinkrug, Efim 163  
Gushcha, Anton V. 72
- Hagedorn, Bastian 179  
Hamfelt, Andreas 306  
Haveraaen, Magne 196  
Holzknecht, Omar 417
- Ivanov, Vladimir 1
- Julliland, Jacques 211
- Kamkin, Alexander 387  
Kärle, Elias 417  
Karpenko, Dmitry 364  
Khoroshilov, Alexey 256
- Klyuchikov, Evgeniy 88  
Konratyev, Dmitry 227  
Kononenko, Irina 147  
Korj, Dmitriy V. 291  
Korovina, Margarita 241  
Kouchnarenko, Olga 211  
Kudinov, Oleg 241
- Lanese, Ivan 95  
Larsen, Stephan Thordal 95
- Makarov, Vladimir 132  
Mallouli, Wissam 59  
Mandrioli, Dino 12  
Mandrykin, Mikhail 256  
Masson, Pierre-Alain 211  
Mazzara, Manuel 1, 95  
McKeever, Steve 306  
Morozov, Sergey 276  
Mustafin, Ruslan 95
- Nardid, Anatoly N. 72  
Nikiforov, Denis A. 291  
Novikov, Evgeny 402
- Paçacı, Görkem 306  
Pak, Vadim 322  
Pavlov, Vladimir 322
- Rodrigues Zalipynis, Ramon Antonio 337  
Rogers, Alan 1  
Rossi, Matteo 12
- Safina, Larisa 95  
Savchenko, Valeriy 352  
Schewe, Klaus-Dieter 117  
Semenov, Vitaly 276  
Shachnev, Dmitry 364  
Sharygin, Eugene 375  
Sher, Arseny 375  
Sidorova, Elena 147  
Sillitti, Alberto 1  
Şimşek, Umutcan 417  
Sivakov, Ruslan L. 291

Smirnov, Kirill 88  
Stephan, Heike 43  
Steuwer, Michel 179  
Succi, Giancarlo 1

Tarlapan, Oleg 276  
Tatarnikov, Andrei 387  
Tec, Loredana 117  
Toussaint, Marcel 394

Vasiliev, Ivan 132  
Vicentini, Federico 12  
Voiron, Guillaume 211  
Volkov, Alexander 352

Zakharov, Ilja 402  
Zhuykov, Roman 375  
Zolotov, Vladislav 276  
Zouev, Eugene 1