Proceedings of the 30^{th} Nordic Workshop on Programming Theory

Daniel S. Fava
Einar B. Johnsen
Olaf Owe
Editors

Report Nr. 485
ISBN 978-82-7368-450-9

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

October 24, 2018

Preface

This volume contains the extended abstracts of the talks presented at the 30^{th} Nordic Workshop on Programming Theory held on October 24-26, 2018 in Oslo, Norway.

The objective of Nordic Workshop on Programming Theory is to bring together researchers from the Nordic and Baltic countries interested in programming theory, in order to improve mutual contacts and co-operation. However, the workshop also attracts researchers outside this geographical area. In particular, it is targeted at early- stage researchers as a friendly meeting where one can present work in progress. Typical topics of the workshop include:

- semantics of programming languages,
- programming language design and programming methodology,
- programming logics,
- formal specification of programs,
- program verification,
- program construction,
- tools for program verification and construction,
- program transformation and refinement,
- real-time and hybrid systems,
- models of concurrency and distributed computing,
- language-based security.

This volume contains 29 extended abstracts of the presentations at the workshop including the abstracts of the three distinguished invited speakers.

Prof. Andrei Sabelfeld, Chalmers University, Sweden
Prof. Erika Ábrahám, RWTH Aachen University, Germany
Prof. Peter Ölveczky, University of Oslo, Norway

After the workshop selected papers will be invited, based on the quality and topic of their presentation at the workshop, for submission to a special issue of *The Journal of Logic and Algebraic Methods in Programming*.

We thankfully acknowledge support from the Norwegian Research Council and the project IoTSec - Security in IoT for Smart Grids, as well as support from the Department of Informatics through the ConSeRNS Strategic Research Initiative for Concurrent Security and Robustness for Networked Systems and the SIRIUS Center for Research-driven Innovation addressing problems of scalable data access in the oil and gas industry.

October 24, 2018
Oslo

Daniel S. Fava
Einar Broch Johnsen
Olaf Owe

Program Committee

Lars Birkedal	Aarhus University
Johannes Borgström	Uppsala University
Daniel S. Fava	University of Oslo
John Gallagher	Roskilde University
Michael R. Hansen	Technical University of Denmark
Magne Haveraaen	University of Bergen
Keijo Heljanko	Aalto University
Fritz Henglein	University of Copenhagen
Thomas Hildebrandt	University of Copenhagen
Anna Ingólfsdóttir	Reykjavík University
Einar Broch Johnsen	University of Oslo
Jaakko Järvi	University of Bergen
Alberto Lluch Lafuente	Technical University of Denmark
Yngve Lamo	Bergen University College
Kim Guldstrand Larsen	Aalborg University
Fabrizio Montesi	University of Southern Denmark
Wojciech Mostowski	Halmstad University
Olaf Owe	University of Oslo
Philipp Ruemmer	Uppsala University
Gerardo Schneider	Chalmers University of Gothenburg
Cristina Seceleanu	Mälardalen University
Jiri Srba	Aalborg University
Tarmo Uustalu	Reykjavik University
Jüri Vain	Tallinn University of Technology
Antti Valmari	University of Jyväskylä
Marina Waldén	Abo Akademi University

Table of Contents

Symbolic Computation Techniques in SMT Solving

Mathematical Beauty meets Efficient Heuristics

Erika Ábrahám

RWTH Aachen University, Germany

Abstract

Checking the satisfiability of quantifier-free real-arithmetic formulas is a practically highly relevant but computationally hard problem. Some beautiful mathematical decision procedures implemented in computer algebra systems are capable of solving such problems, however, they were developed for more general tasks like quantifier elimination, therefore their applicability to satisfiability checking is often restricted. In computer science, recent advances in satisfiability-modulo-theories (SMT) solving led to elegant embeddings of such decision procedures in SMT solvers in a way that combines the strengths of symbolic computation methods and heuristic-driven search techniques. In this talk we discuss such embeddings and show that they might be quite challenging but can lead to powerful synergies and open new lines of research.

Design and Validation of Cloud Storage Systems using Rewriting Logic

Peter Ölveczky

University of Oslo, Norway

Abstract

To deal with large amounts of data while offering high availability and throughput and low latency, cloud computing systems rely on distributed, partitioned, and replicated data stores. Such cloud storage systems are complex software artifacts that are very hard to design and analyze. We argue that formal specification and model checking should be beneficial during their design and validation. In particular, I propose rewriting logic and its accompanying Maude tools as a suitable framework for formally specifying and analyzing both the correctness and the performance of cloud storage systems. This talk gives an overview of the use of rewriting logic at the University of Illinois' Assured Cloud Computing center on industrial data stores such as Google's Megastore and Facebook/Apache's Cassandra. I also briefly summarize the experiences of the use of a different formal method for similar purposes by engineers at Amazon Web Services.

Breaking and Fixing IoT Apps
From Attacks to Formal Methods for Security

Andrei Sabelfeld*

Chalmers University, Sweden

Abstract

IoT apps empower users by connecting a variety of otherwise unconnected services. Unfortunately, the power of IoT apps can be abused by malicious makers, unnoticeably to users. We demonstrate that popular IoT app platforms are susceptible to several classes of attacks that violate user privacy, integrity, and availability. We estimate the impact of these attacks by an empirical study. We suggest short/medium-term countermeasures based on fine-grained access control and long-term countermeasures based on information flow tracking. Finally, we discuss general trends and challenges for the Web of Things and, in particular, the role of formal methods in securing it.

*Parts of the talk are based on joint work with Iulia Bastys and Musard Balliu.

A verified proof checker for higher-order logic

Oskar Abrahamsson[1]

Chalmers University of Technology, Sweden
aboskar@chalmers.se

Abstract

We present a verified mechanized checker for proofs in higher-order logic (HOL). The proof checker is implemented in CakeML, and utilizes the Candle theorem prover kernel to check logical inferences. The checker reads proofs in the OpenTheory article format, meaning that proofs produced by several commonly used HOL proof assistants are supported. The proof checker is implemented and verified using the HOL4 theorem prover, and comes with a proof of soundness.

1 Introduction

In this work, we present a verified proof checker for proofs in higher-order logic. The proof checker is implemented in effectful CakeML, which is synthesized from a monadic HOL specification using a proof-producing synthesis mechanism [2]. The checker operates on proofs in the OpenTheory article format, and utilizes the verified Candle theorem prover kernel [4] to perform all logical inferences.

We verify the correctness of the proof checker, and prove a soundness theorem. This theorem guarantees that any theorem produced as a result of a successful run of the tool is a theorem in HOL. Together with the end-to-end correctness theorem of the CakeML compiler [6] this implies that the proof checker is sound down to the machine code which executes it.

We start by providing some background information on the techniques used in the implementation and verification of the checker (§2). We then provide a high-level overview of how the proof checker is implemented (§3), and state a theorem about the soundness of the proof checker, and explain, at a high level, how the soundness theorem is proven using the existing soundness theorem of the Candle kernel (§4). Finally, we comment on running the checker on article files (§5).

2 Background

Our efforts to produce a verified proof checker are made possible largely by the following results.

(i) The OpenTheory framework enables proof recording in several HOL ITP systems, and provides an abstract stack machine for replaying such proofs. We construct a logical specification of the stack machine in monadic HOL.

(ii) The Candle theorem prover kernel is used to perform logical inferences within the OpenTheory framework, and its soundness result is used as a basis for the soundness result of the proof checker.

(iii) The CakeML compiler ecosystem is used to synthesize effectful CakeML code from the monadic specification of the OpenTheory article checker. The verified compiler is also used to compile the verified CakeML code to concrete machine code within HOL4.

OpenTheory The OpenTheory framework [3] provides a means for sharing logical theories between interactive theorem provers (ITPs) that use HOL as their logic. An OpenTheory article checker reads in a text file consisting of low-level operations on a stack machine where the operations are primitive inferences and term constructors/destructors of HOL.

Candle The Candle theorem prover kernel is a verified implementation of HOL light in HOL4 by Kumar et al. [4]. The kernel is implemented as monadic HOL functions, and is proven sound with respect to a formal semantics which builds on Harrison's formalization of HOL light [1].

CakeML CakeML is a language in the Standard ML family of functional programming languages. The CakeML language has a formal semantics, and supports most features present in Standard ML, such as references, I/O, and exceptions. The CakeML ecosystem consists of the CakeML language, a fully verified compiler which is able to bootstrap itself inside the logic, and a proof-producing synthesis mechanism which is able to synthesize (effectful) CakeML functions from (monadic) HOL functions.

3 Approach

Here is a high-level overview of the steps taken to achieve our result of a verified proof checker.

(i) We begin by specifying the OpenTheory stack machine in HOL4 as monadic functions. The stack machine performs bookkeeping of theorems, constants and types, and calls on the Candle kernel to perform logical inferences.

(ii) We synthesize semantically equivalent CakeML code from the specification in step (i) using the proof-producing mechanism described in Ho et al. [2].

(iii) We prove a series of correctness results for the OpenTheory proof checker. Using the existing Candle soundness theorem, we prove that any valid sequent produced by a successful run of the proof checker is in fact true by the semantics of HOL.

(iv) The CakeML compiler is used to compile the stateful CakeML code from (ii) to executable machine code. The compilation is carried out wholly within the HOL4 logic, and produces a theorem that the resulting machine code satisfies the specification from (iii).

4 End-to-end correctness

In this section, we explain one of the main soundness theorems that we have proved for the proof checker. The theorem is stated in terms of the specification of the article reader, reader_main. Here is the theorem:

$$
\begin{array}{l}
\vdash \mathsf{is\_set\_theory}\ \mu \Rightarrow \\
\quad \mathsf{reader\_main}\ \mathit{fs}\ \mathsf{init\_refs}\ \mathit{cl} = (\mathsf{T}, \mathit{outp}, \mathit{refs}, \mathit{fstate}) \Rightarrow \\
\quad \exists\, s. \\
\qquad \mathit{fstate} = \mathsf{SOME}\ s \wedge \\
\qquad (\forall\, \mathit{asl}\ c.\ \mathsf{mem}\ (\mathsf{Sequent}\ \mathit{asl}\ c)\ s.\mathsf{thms} \Rightarrow (\mathsf{thyof}\ \mathit{refs}.\mathsf{the\_context}, \mathit{asl}) \models c) \wedge \\
\qquad \mathit{refs}.\mathsf{the\_context}\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge \\
\qquad \mathit{outp} = \mathsf{add\_stdout}\ \mathit{fs}\ (\mathsf{msg\_success}\ s\ \mathit{refs}.\mathsf{the\_context})
\end{array}
$$

The key points of the theorem are the following:

- The predicate is_set_theory assumes the existence of a powerful enough set theory to express the semantics of HOL light (see [4] for a discussion). This assumption is required to replace a syntactic entailment in the theorem by a semantic entailment.
- The HOL function reader_main is the main specification of the proof checker. It takes as input a model of a UNIX file system, an initial state for the Candle kernel, and the command-line arguments with which the proof checker was called.
- The Boolean value T implies that (a) the proof checker was given a valid input file, and (b) it processed all input without printing an error message.
- The string *outp* is the output printed by the checker to stdio. Here, msg_success is a message containing the *context* (i.e. constant- and type definitions, and axioms) of the Candle kernel final state, and all sequents constructed by the proof checker (residing in the stack machine post-state s).
- Finally, all sequents constructed by the proof checker during a successful run will reside in the OpenTheory stack machine post-state s, and these sequents are actually true by the semantics of HOL. Moreover, the context *refs*.the_context is the result of a series of valid updates of the initial context.

The theorem shown above can be connected to the compiler correctness theorem (see §3), and thus become a statement about the generated machine code, since the CakeML compiler can be run inside the HOL4 logic. See Kumar et al. [5] for a discussion.

5 Results

Our proof checker has been used to check some articles from the OpenTheory standard library. All checked theories were successfully processed without errors. When compared against the OpenTheory toolset [3], our verified checker runs a factor of 7 times slower on average. A significant portion of this slowdown is caused by poor I/O performance.

Acknowledgments. The original implementation of the OpenTheory stack machine in monadic HOL was done by Ramana Kumar, who also provided helpful support during the course of this work. The author would also like to thank Magnus Myreen for feedback on this text. Finally, the author thanks the anonymous reviewers for their helpful comments.

References

[1] John Harrison. Towards self-verification of HOL light. In *IJCAR*, pages 177–191, 2006.

[2] Son Ho, Oskar Abrahamsson, Ramana Kumar, Magnus O. Myreen, Yong Kiam Tan, and Michael Norrish. Proof-producing synthesis of CakeML with I/O and local state from monadic HOL functions. In *IJCAR*, pages 646–662, 2018.

[3] Joe Hurd. The OpenTheory standard theory library. In *NFM*, pages 177–191, 2011.

[4] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-formalisation of higher-order logic - semantics, soundness, and a verified implementation. *JAR*, 56(3):221–259, 2016.

[5] Ramana Kumar, Eric Mullen, Zachary Tatlock, and Magnus O. Myreen. Software verification with ITPs should use binary code extraction to reduce the TCB. In *ITP*, pages 362–369, 2018.

[6] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *POPL*, pages 179–192, 2014.

The Complexity of Identifying Characteristic Formulae*†

Luca Aceto[1,2], Antonis Achilleos[1], Adrian Francalanza[3], and Anna Ingólfsdóttir[1]

[1] School of Computer Science, Reykjavik University, Reykjavik, Iceland
[2] Gran Sasso Science Institute, L'Aquila, Italy
[3] Dept. of Computer Science, ICT, University of Malta, Msida, Malta

Abstract

We examine the complexity of determining whether a modal formula (possibly with recursion operators) characterizes a process up to bisimulation equivalence.

1 Introduction

Characteristic formulae are formulae that characterize a process up to some notion of behavioural equivalence or preorder, which in our case is bisimilarity: a formula φ is characteristic for a process p when every process q is bisimilar to p exactly when it satisfies φ. A construction of characteristic formulae for variants of CCS processes [9] was introduced in [6]. This construction allows one to verify that two CCS processes are equivalent by reducing this problem to model checking. Similar constructions were studied later in, for instance, [1, 10, 12].

We are interested in detecting when a formula is characteristic for a certain process. We call this the characterization problem and we determine its complexity. We focus on a representative collection of logics, including a selection of modal logics without recursion and maxHML, the max-fragment of μHML [8], a variant of the μ-calculus [7], which consists of the μHML formulae that only use greatest fixed points. These formulae are sufficient to provide characteristic formulae for any state in a finite labelled transition system.

Similar to the characterization problem is the completeness problem, which asks whether a given formula is complete, meaning that any two processes that satisfy it are bisimilar to each other. Therefore, a complete formula is characteristic if and only if it is satisfiable. As we see in the following sections, the completeness problem tends to have the same complexity as validity. The techniques that we use to determine the complexity of completeness for modal logics without fixed points were presented in [2], but we extend these techniques for the case of maxHML and show that the completeness problem for this fragment is EXP-complete. The EXP-completeness of the characterization problem is an immediate corollary.

2 Background

Definition 1. *The formulae of that we consider are constructed using the following grammar:*

$$\varphi, \psi ::= p \mid \neg p \mid \mathit{tt} \mid \mathit{ff} \mid X \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle \alpha \rangle \varphi \mid [\alpha]\varphi \mid \mu X.\varphi \mid \nu X.\varphi$$

where X comes from a countably infinite set of logical variables LVar*, α from a finite set of actions,* Act*, and p from a countable set of propositional variables, P.*

*This extended abstract is partly based on results from [2].

†This research was supported by the project "TheoFoMon: Theoretical Foundations for Monitorability" (grant number: 163406-051) and the project "Epistemic Logic for Distributed Runtime Monitoring" (grant number: 184940-051) of the Icelandic Research Fund.

We interpret formulae on the states of a labelled transition system (LTS). An LTS is a quadruple $\langle \textsc{Proc}, \textsc{Act}, \rightarrow, V \rangle$ where $\textsc{Proc}$ is a set of states or processes, $\textsc{Act}$ is the set of actions, $\rightarrow \subseteq \textsc{Proc} \times \textsc{Act} \times \textsc{Proc}$ is a transition relation, and $V : P \rightarrow 2^{\textsc{Proc}}$ determines on which states a propositional variable is true. We assume that our LTS contains all the possible finite behaviours and only those. State `nil` represents any state that cannot transition anywhere: $\forall \alpha \forall s. \texttt{nil} \not\xrightarrow{\alpha} s$. The size of a state s is $|s|$, the number of states that can be reached from s by any sequence of transitions, and $|\varphi|$ is the length of φ as a string of symbols. All our complexity results are with respect to these measures.

Formulae are evaluated in the context of an LTS and an environment, $\rho : \textsc{LVar} \rightarrow 2^{\textsc{Proc}}$, which gives values to the logical variables. For an environment ρ, variable X, and set $S \subseteq \textsc{Proc}$, $\rho[X \mapsto S]$ is the environment which maps X to S and all $Y \neq X$ to $\rho(Y)$. The semantics for our formulae is given through a function $[\![\cdot]\!]$:

$$[\![\texttt{tt}, \rho]\!] = \textsc{Proc}, \quad [\![\texttt{ff}, \rho]\!] = \emptyset, \quad [\![p, \rho]\!] = V(p), \quad [\![\neg p, \rho]\!] = \textsc{Proc} \setminus V(p), \quad [\![X, \rho]\!] = \rho(X)$$

$$[\![\varphi_1 \wedge \varphi_2, \rho]\!] = [\![\varphi_1, \rho]\!] \cap [\![\varphi_2, \rho]\!] \qquad [\![[\alpha]\varphi, \rho]\!] = \left\{ s \mid \forall t.\ s \xrightarrow{\alpha} t \text{ implies } t \in [\![\varphi, \rho]\!] \right\}$$

$$[\![\varphi_1 \vee \varphi_2, \rho]\!] = [\![\varphi_1, \rho]\!] \cup [\![\varphi_2, \rho]\!] \qquad [\![\langle \alpha \rangle \varphi, \rho]\!] = \left\{ s \mid \exists t.\ s \xrightarrow{\alpha} t \text{ and } t \in [\![\varphi, \rho]\!] \right\}$$

$$[\![\nu X.\varphi, \rho]\!] = \bigcup \{ S \mid S \subseteq [\![\varphi, \rho[X \mapsto S]]\!] \} \quad [\![\mu X.\varphi, \rho]\!] = \bigcap \{ S \mid S \supseteq [\![\varphi, \rho[X \mapsto S]]\!] \}$$

A formula is closed when every occurrence of a variable X is in the scope of recursive operator νX or μX. Henceforth we consider only closed formulae. As the environment has no effect on the semantics of a closed formula φ, we write $s \models \varphi$ for $s \in [\![\varphi, \rho]\!]$. Depending on how we further restrict our syntax, and the LTS, we can describe several logics. Without further restrictions, the resulting logic is the μ-calculus. If we do not allow any propositional variables, the resulting logic is μHML, and if we further disallow the recursive operators, the resulting logic is HML. If we allow propositional variables, but only one action and no recursive operators (or recursion variables), then we have the basic modal logic **K**, and further restrictions on the LTS can result in a wide variety of modal logics. For example, logic **D** has all the restrictions of **K**, but furthermore, `nil` is not allowed as a state, while for logic **S4**, the transition relation must be reflexive and transitive — for more on Modal Logic, see [3, 4]. For convenience and brevity, we examine the logics HML, $\mathbf{D}^-$ that has all the restrictions of both HML and **D**, and maxHML that allows only the formulae of μHML that do not use the operator μX. For more details on our techniques — mostly on non-recursive logics — the reader can see [2].

In the context of these logics, we call a formula φ *characteristic* for state s when $s \models \varphi$ and for every state t, $s \sim t$ if and only if $t \models \varphi$, where $\sim$ stands for bisimilarity without accounting for variables. The characterization problem is the following: *Given a formula φ and a state s, is φ characteristic for s?* A formula φ is called *complete* when for all states s and t, if $s \models \varphi$ and $t \models \varphi$, then $s \sim t$. The completeness problem is: *Given a formula φ, is φ complete?*

3 The Complexity of Completenes and Characterization

Proposition 1. *The completeness and characterization problems for $\boldsymbol{D}^-$ are in P.*

Proof. For $\mathbf{D}^-$, all states are bisimilar, so all formulae are complete; for more, see [2]. □

It is known that satisfiability for the min-fragment of the μ-calculus (on one action) is EXP-complete. It is in EXP, as so is the satisfiability problem of the μ-calculus [7]. Furthermore, this fragment suffices [11] to describe the PDL formula that is constructed by the reduction

used in [5] to prove EXP-hardness for PDL, therefore the reduction can be adjusted to prove that the min-fragment of the μ-calculus is EXP-complete. Therefore, validity for the min- and max-fragments of the μ-calculus (on one action) is EXP-complete. To see that this lower bound transfers to minHML and maxHML, it suffices to use one extra action to represent propositional variables (for example, variable x_i can be replaced by $\langle\alpha\rangle^i\mathtt{tt}$).

Proposition 2. *The completeness problems for HML and* maxHML *are* PSPACE*-hard and* EXP*-hard, respectively.*

Proof. We prove the theorem for the case of μHML by a reduction from minHML-validity. First, notice that $\bigwedge_{\alpha\in\text{Act}}[\alpha]\mathtt{ff}$ is complete and it is satisfiable by process `nil`. Given a minHML-formula φ, there are two cases. If $\mathtt{nil} \models \neg\varphi$, then φ is not valid and we set $\varphi_c = \mathtt{tt}$. Otherwise, let $\varphi_c = \neg\varphi \vee \bigwedge_{\alpha\in\text{Act}}[\alpha]\mathtt{ff}$. For the second case, if φ is valid, then φ_c is equivalent to $\bigwedge_{\alpha\in\text{Act}}[\alpha]\mathtt{ff}$, which is complete; if φ_c is complete, then only `nil` satisfies it and therefore, φ is valid. Thus, in both cases, φ is valid if and only if φ_c is complete. □

Theorem 3. *The completeness problems for HML and* maxHML *are* PSPACE*-complete and* EXP*-complete, respectively.*

Notice that in the proof of Proposition 2, the reduction could have returned both φ_c and `nil`, instead of just φ_c. Furthermore, as model-checking has a lower complexity than completeness, checking if φ is characteristic of s is at most as hard as checking if φ is complete. Therefore, the same complexity bounds hold for the characterization problem of HML and maxHML.

References

[1] Luca Aceto, Anna Ingolfsdottir, Paul Blain Levy, and Joshua Sack. Characteristic formulae for fixed-point semantics: a general framework. *Mathematical Structures in Computer Science*, 22(02):125–173, 2012.

[2] Antonis Achilleos. The completeness problem for modal logic. In *Logical Foundations of Computer Science*, Lecture Notes in Computer Science, pages 1–21. Springer, 2018.

[3] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[4] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. The MIT Press, 1995.

[5] Michael J Fischer and Richard E Ladner. Propositional dynamic logic of regular programs. *Journal of computer and system sciences*, 18(2):194–211, 1979.

[6] S. Graf and J. Sifakis. A modal characterization of observational congruence on finite terms of CCS. *Information and Control*, 68(1-3):125–145, January 1986.

[7] Dexter Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.

[8] Kim Guldstrand Larsen. Proof systems for satisfiability in Hennessy-Milner logic with recursion. *Theoretical Computer Science*, 72(2&3):265–288, 1990.

[9] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[10] Markus Mller-Olm. Derivation of characteristic formulae. *Electronic Notes in Theoretical Computer Science*, 18:159–170, 1998.

[11] V. R. Pratt. A decidable mu-calculus: Preliminary report. In *22nd Annual Symposium on Foundations of Computer Science (sfcs 1981)*. IEEE, oct 1981.

[12] Bernhard Steffen and Anna Ingólfsdóttir. Characteristic formulas for processes with divergence. *Information and Computation*, 110(1):149–163, 1994.

A Runtime Enforcement Mechanism for Collaborative Privacy Policies in Online Social Networks

Hanaa Alshareef[1], Raúl Pardo[2] and Gerardo Schneider[1]

[1] Department of Computer Science and Engineering,
Chalmers | University of Gothenburg, Sweden.
[2] INRIA, Lyon, France.
hanaa@chalmers.se, raul.pardo-jimenez@inria.fr, gersch@chalmers.se

1 Motivation

Most Online Social Networks (OSNs) today have privacy settings that allow users to define their preferences in what concerns the use of their data. This usually includes aspects related to whom can have access to which information but it is limited in a number of ways. For example, OSNs such as Facebook and Instagram may only describe who is the direct audience of a given item (post, message, picture, etc.), meaning that it only concerns who have access to the item based on the explicit relationships the user has previously defined. In many cases it involves only one level in the relationship order (e.g., friends) or two levels (e.g., friends of friends). Additionally, users might be interested in defining privacy policies that limit the access to other users not directly connected with us beyond two levels, as it is the case whenever somebody who originally got access to the information wants to (re)share it.

Another issue is who decides whom has access to the information. Today, in the majority of the OSNs the owner, who has the piece of information in her profile, is the one solely defining and regulating the privacy settings. Many other users are also concerned with the posted item, so they should also have a say in who may access to it or not. Ideally, there should be a mechanism allowing all the involved users to take a decision collaboratively. Current implementations of social networks do not allow a fine grained enforcement in case the posted item concerns many users, and the privacy settings usually do not allow for setting limits when a user wants to share the item got access to. Moreover, it is quite common that the involved users' privacy settings are in contradiction so there is a need to solve the conflicts before deciding who has access to the shared object.

While OSNs are considered to be a collaborative environment where most of activities involve some users, current access control mechanisms suffers from collaborative policy limitations. This lack of collaborative policies for access control violates the privacy of all the users who share a particular content with the owner by delegating the full responsibility over their privacy settings to the owner.

The main contribution of this paper is a framework to the problem of posting and sharing items in OSNs whenever multiple users are involved. Additionally,we have deployed our solution and provide a proof-of-concept implementation using [4] which is an open source OSN.

2 A Collaborative Access Control Framework for OSNs

We briefly introduce here the components of our framework: OSN model, privacy policy specification, requirements for a conflict resolution strategy and the algorithms for collaborative decisions.

2.1 OSN Model

OSNs are typically structured as graphs, where vertices represent users and items and the edges of the graph represent connections between nodes. Concretely, vertices in the model are split into users, items,

and groups. Relationship types represent connections between vertices in the graph. For example, social relationships between users such as friends, colleagues, family and so forth; or relationships between users and items or groups such as Alice is the owner of post p, Bob is mentioned in p, Carol owns group g, or David is a member in g. For simplicity in the rest of the paper and without loss of generality we assume that there can only exist one relationship between any two vertices in the graph. Similarly to [6], we define a set of *controllers types*:

Owner. An owner is a user who owns all items located in her profile.
Stakeholder. A stakeholder is a user who is tagged or mentioned in an item.
Contributor. A contributor is a user that posts an item in a profile different than hers.
Originator. A user is considered to be an originator when an item is shared from her profile.

2.2 Setting of our Collaborative Privacy Policy Framework

1 (The Privacy Policy Specification) This specification contains access specification, data specification and access control policies. In our framework each controller can identify a set of users who can access her data and who cannot, the so-called *accessors*. The controller can specify her permitted and denied accessors by three parameters: *user names*, *group names* and *relationship types*. These accessor types help the controller to customize her access control policy. Regarding data specification, our model focuses on the users item and assigns a level of sensitivity to the item based on how much a disclosure would harm the user. Sensitivity levels of shared items help to effectively solve conflicts between controllers. Finally, the access control policies of a given item contain the policy of each associated controller who has relationship with the item.

2 (Requirements for A Conflict Resolution Strategy) Inferring fuzzy trust, controllers weight scheme and accessors' weight scheme are the principles that we use to combine the different individual privacy policies (i.e., the conflict resolution strategy). Users explicitly identify a trust value for those with whom they have a direct relationship expressed in a *trust graph*. When users are not directly connected, we adopt the Lesani and Bagheri [7] approach to calculate the trust between two users. *Controllers' weight scheme* is a method to determine priorities and impact levels of a controller's access control policy. We use the principle that people who are close to each other tend to be similar [1, 5], to weight each controller. In our framework not all accessors are equal because they are specified differently; thus, we weight the accessors based on how they are authorized or denied. In response to this assumption, we adopt the *most-specific-takes-precedence* principle to weight our accessors [2, 3].

2.3 Algorithms of Collaborative Privacy Decisions

PermittedandDeniedAccessors and *Sharing* are the main algorithms behind our collaborative privacy management of posted and shared items. The *PermittedandDeniedAccessors* produces a final set of accessors who are permitted to view the item -the so-called *viewers*- and those who are denied. When a viewer (or controller) is granted a permission to share the item with her social network, she is then called the *disseminator*. The *Sharing* algorithm produces sets of users who are allowed to disseminate the item and who are not. We do not claim that the algorithms we present are the right ones, as different decision could be taken depending on whether one might want to privilege privacy over utility or *vice-versa*. This trade-off between privacy and utility may be stretched or relaxed by playing with the weights that may be associated to different aspects of our decision algorithms.

2.4 Implementation

We have implemented[2] our solution in the open source social network Diaspora [4], and we have tested the system by creating a complex social graph to show the most interesting features that our approach can address. Note that currently Diaspora does not offer any privacy settings, so our work could eventually have great impact in practice if our solution is adopted by the community.

3 Acknowledgments

This research has been supported by Culture Bureau of the Embassy of the Kingdom of Saudi Arabia in Berlin.

References

[1] K. Carley. A theory of group stability. *American sociological review*, pages 331–354, 1991.

[2] S. De Capitani Di Vimercati, S. Foresti, P. Samarati, and S. Jajodia. Access control policies and languages. *International Journal of Computational Science and Engineering*, 3(2):94–102, 2007.

[3] S. D. C. di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. In *International Workshop on Databases in Networked Information Systems*, pages 225–237. Springer, 2005.

[4] Diaspora. Diaspora. `https://joindiaspora.com`, 2018. [Available Online].

[5] S. L. Feld. The focused organization of social ties. *American journal of sociology*, 86(5):1015–1035, 1981.

[6] H. Hu, G.-J. Ahn, and J. Jorgensen. Multiparty access control for online social networks: model and mechanisms. *IEEE Transactions on Knowledge and Data Engineering*, 25(7):1614–1627, 2013.

[7] M. Lesani and S. Bagheri. Applying and inferring fuzzy trust in semantic web social networks. In *Canadian Semantic Web*, pages 23–43. Springer, 2006.

[2] https://github.com/alshareef-hanaa/collaborative-model

A Survey of Semantic Attribute-Based Access Control Schemes

Hamed Arshad

Department of Informatics, University of Oslo, Norway. hamedar@ifi.uio.no

Abstract

Attribute-based access control (ABAC) has several advantages over the traditional access control models such as the mandatory access control (MAC), discretionary access control (DAC), role-based access control (RBAC), and so on. ABAC uses the attributes of the involving entities (i.e., subjects, objects, environments, actions) to provide the access control. Despite various advantages offered by ABAC, it is not the best fit for distributed and heterogeneous environments, where the attributes of an entity do not necessarily match (syntactically) those used in the access policies. Therefore, another type of access control called semantic attribute-based access control (SABAC) has emerged that tries to associate syntactically different but semantically equivalent attributes in different domains by combining the semantic technologies with the ABAC. Over the last decade, a number of research efforts have been conducted in developing semantic attribute-based access control schemes. However, there exists no survey paper on SABAC schemes giving an overview of the existing approaches. Hence, this paper comprehensively reviews the conducted research efforts for developing SABAC. The main goal of this paper is to provide a comprehensive summary of the conducted research studies that is useful for researchers who want to enter and make contribution to this field. The paper identifies the open problems and possible research entry points by demonstrating the advantages and disadvantages of the previous works.

Attribute-based access control (ABAC) is a successor of the role-based access control (RBAC), where involving entities (e.g., subject, object, action, and environment) have some attributes and the access control is provided based on these attributes. Under the ABAC model, there is no need to assign capabilities to subjects (e.g., users, groups, roles, etc.) in advance. Receiving an access request, the access decision would be made based on the attributes of the requested object (resource), attributes of the requester (subject), conditions of the environment (e.g., time of the day, authentication level, location, etc.), attributes of the desired action, and predefined access control policies. ABAC has several advantages over the traditional access control models such as MAC, DAC, RBAC, and so on. ABAC has reached the maturity of OASIS standards with XACML 3.0 (eXtensible Access Control Markup Language, version 3.0) and SAML 2.0. The XACML standard provides a policy language, which is sufficiently fine-grained and declarative, as well as an architecture for ABAC. The standard also specifies the process by which the requests are evaluated based on the defined policies.

ABAC is supposed to be a proper solution in open and distributed systems. However, since such systems are heterogeneous, the attributes of a requester (subject) may not necessarily match those specified in the policies defined for accessing services or data (objects). For example, an e-healthcare system may represent adult patients with an attribute *"Adult"*, while patients may want to demonstrate this by providing something like a license (e.g., by an attribute *"hasDriverLicense"*) or an attribute *"age"*. In a typical attribute-based access control mechanism, this issue should be considered when defining a policy, which in turn makes the management of policies a very complicated and error-prone task. In other words, the policy administrator needs to consider all the possible synonyms (semantically) of each attribute by defining several policies (for the same object) or one general policy. Consequently, when a

change occurs in a policy, a large number of policies need to be updated accordingly. Therefore, it can be concluded that ABAC needs to be extended to be suitable for heterogeneous environments. In order to address such problems, another type of access control called the semantic attribute-based access control (SABAC) is proposed as an extension of ABAC. The idea behind the SABAC is the combination of ABAC with the semantic technologies. In other words, the aim of SABAC schemes is making decisions semantically as well as considering the semantic relationships for inferring new policies (i.e., implicit policies) from the defined policies (explicit ones). Hence, by specifying the access polices at a conceptual level and making decisions with the help of the semantic inference, the access control mechanism can be improved significantly.

In ABAC (and SABAC), different entities are involved in controlling access. These entities (i.e., the subject, object, action, and environment) and their attributes and relationships can be defined formally and in a format that is readable by machine, using an ontology. An ontology can be created using ontology markup languages such as DAML+OIL [9], RDF [14], OWL [2], and RDF Schema (RDFS) [4]. However, if some specific relations need to be held under some conditions, then it is difficult to express them using these ontology markup languages. Such an issue can be handled using the rules. Hence, several rule markup languages, such as XRML [15], SRML [19], RuleML [3], and SWRL [10] have been proposed for this purpose. Most of the existing SABAC schemes use the SWRL, which combines the Horn logic rules and OWL ontologies, as the rule markup language. It is a popular rule markup language as most of reasoners and rule engines (e.g., Pellet [18], Drools [17], Jess [8], Protégé [16], etc.) support it. Therefore, when we talk about the semantic technologies, it means ontologies plus rules.

In order to specify the access policies, we need policy languages. Several policy languages such as XACML [1], XRBAC [11], and Ponder [5] have been proposed for the specification of access policies. However, they do not consider semantics and thus are not suitable for heterogeneous environments, where different systems and domains may use different terminologies. For heterogeneous environments, another type of policy languages (such as KAoS [20], Rei [13], Rein [12], XACML+OWL [7], EXAM-S [6], etc.) have emerged that are based on semantic technologies. However, most of them work based on the knowledge represented by ontologies while complex non-monotonic rules cannot be represented by an ontology. We have found that most of the existing SABAC schemes use the XACML policy language along with a rule markup language like SWRL.

Over the last decade, a number of semantic attribute-based access control schemes have been proposed for different contexts. However, there exists no survey paper on semantic attribute-based access control (SABAC) schemes giving an overview of the existing approaches. Therefore, it is difficult to get knowledge about the existing techniques for SABAC and the advantages/disadvantages of applying each technique. In order to provide a comprehensive reference on SABAC identifying the benefits/drawbacks of the existing approaches and the trends and gaps in this field of study, this paper reviews all the conducted research studies on SABAC systematically and in depth.

Acknowledgements

I would like to thank my supervisors **Dr. Christian Johansen** and **Prof. Olaf Owe** for sharing their pearls of wisdom with me during the course of this research.

References

[1] A. Anderson, A. Nadalin, B. Parducci, D. Engovatov, H. Lockhart, M. Kudo, P. Humenn, S. Godik, S. Anderson, S. Crocker, et al. extensible access control markup language (xacml) version 1.0. *OASIS*, 2003.

[2] S. Bechhofer. Owl: Web ontology language. In *Encyclopedia of database systems*, pages 2008–2009. Springer, 2009.

[3] H. Boley, A. Paschke, and O. Shafiq. Ruleml 1.0: the overarching specification of web rules. In *International Workshop on Rules and Rule Markup Languages for the Semantic Web*, pages 162–178. Springer, 2010.

[4] D. Brickley, R. V. Guha, and B. McBride. Rdf schema 1.1. *W3C recommendation*, 25:2004–2014, 2014.

[5] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In *Policies for Distributed Systems and Networks*, pages 18–38. Springer, 2001.

[6] R. Ferrini. *EXAM-S: an Analysis tool for Multi-Domain Policy Sets*. PhD thesis, alma, 2009.

[7] R. Ferrini and E. Bertino. Supporting rbac with xacml+ owl. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 145–154. ACM, 2009.

[8] E. F. Hill. *Jess in action: Java rule-based systems*. Manning Publications Co., 2003.

[9] I. Horrocks et al. Daml+oil: A description logic for the semantic web. *IEEE Data Eng. Bull.*, 25(1):4–9, 2002.

[10] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21:79, 2004.

[11] J. B. Joshi. Access-control language for multidomain environments. *IEEE Internet Computing*, 8(6):40–50, 2004.

[12] L. Kagal, T. Berners-Lee, D. Connolly, and D. Weitzner. Using semantic web technologies for policy management on the web. In *Proceedings of the national conference on Artificial Intelligence*, volume 21, page 1337. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[13] L. Kagal, T. Finin, and A. Joshi. A policy language for a pervasive computing environment. In *Policies for Distributed Systems and Networks, 2003. Proceedings. POLICY 2003. IEEE 4th International Workshop on*, pages 63–74. IEEE, 2003.

[14] G. Klyne and J. J. Carroll. Resource description framework (rdf): Concepts and abstract syntax. 2006.

[15] J. K. Lee and M. M. Sohn. The extensible rule markup language. *Communications of the ACM*, 46(5):59–64, 2003.

[16] M. A. Musen. Protégé ontology editor. *Encyclopedia of Systems Biology*, pages 1763–1765, 2013.

[17] M. Proctor. Drools: a rule engine for complex event processing. In *Proceedings of the 4th international conference on Applications of Graph Transformations with Industrial Relevance*, pages 2–2. Springer-Verlag, 2011.

[18] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[19] M. Thorpe and C. Ke. Simple rule markup language (srml): a general xml rule representation for forward-chaining rules. *XML coverpages*, page 1, 2001.

[20] A. Uszok, J. M. Bradshaw, M. Johnson, R. Jeffers, A. Tate, J. Dalton, and S. Aitken. Kaos policy management for semantic web services. *IEEE Intelligent Systems*, 19(4):32–41, 2004.

Adaptations of API test case generation

Cyrille Artho[1], Rudolf Ramler[2], Martina Seidl[3], and Jun Yoneyama[4]

[1] KTH Royal Institute of Technology, Stockholm, Sweden
artho@kth.se
[2] Software Competence Center Hagenberg, Hagenberg, Austria
rudolf.ramler@scch.at
[3] Johannes Kepler University, Linz, Austria
martina.seidl@jku.at
[4] The University of Tokyo, Tokyo, Japan
yone_j_synthesis@yahoo.co.jp

Abstract

Online test case generation directly executes the system under test, by calling functions of its application programming interface (API). Such direct API testing can be extended by calling an adapter layer instead of the system. This unifies offline and online testing and also opens the door to different types of applications of test case generation. These include generation data structures instead of code, interfacing with systems that do not have an API, including systems with a graphical user interface, and even model-based simulation. We give a new view on this problem by looking at our past work.

1 Introduction

Model-based testing derives concrete test cases from an abstract test model [7, 11]. Test derivation techniques can be divided into online testing and offline testing [11]. In online testing, the test generation (derivation) tool connects directly to the system under test (SUT) to execute test cases. In offline test generation, test derivation tool first generates an intermediate representation of test cases that is turned into executable tests later [11].

In online testing, test sequences generated by the model are often direct function calls to an application programming interface (API) with concrete data. If test cases are generated offline, or not in a format that can be directly used to invoke the SUT, they first have to be *adapted* to the SUT, by using the Adapter design pattern [8]. An adapter is an intermediate software layer that converts abstract inputs and outputs from and to a test tool into the right form for the SUT [1] (see Figure 1). Adaptation can be a time-consuming task [12]. However, it also provides an opportunity to extend the use cases of model-based test tools to new types of applications. Four such cases are presented in this paper.

This paper is based on our experience, but covers generic techniques. Past results were obtained using tools Modbat [2, 5] and Randoop [9], but generalize to other test case generation tools. We show that test adaptation, often seen as a burden [12], allows us to interface with systems that are otherwise not easily amenable to test automation.

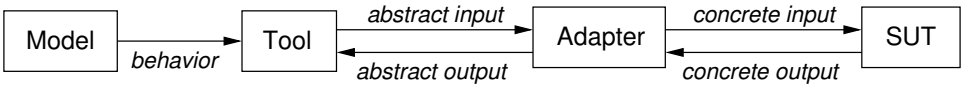


Figure 1: Model-based testing with an adapter.

Adapter type	Implementation sketch	Role of adapter
Data structure	`data = new DataObject(...); data.addItem(...);`	Creation of data in the right format
API-less system	`connection.send("GET /index.html HTTP/1.1");`	Interaction via specific protocol
GUI	`robot.mouseMove(...); leftClick();`	Interface to system-specific GUI
Simulation	`environment.setDelay(...);`	Simulation orchestration and control

Table 1: Examples of different adapter types, shown in pseudo code.

2 Adaptations

Table 1 illustrates the different adapter types with code snippets that implement actions as concrete commands that the system can use. We explain each case in the following.

2.1 Data structure generation

Satisfiability (SAT) solvers are the core reasoning engines of many verification systems. Hence, such solvers have to be trustworthy, despite themselves being very advanced pieces of software. We empirically showed that model-based testing is extremely useful for building reliable solvers [6], using Modbat to test incremental SAT solvers [5]. To generate test data, random input formulas are passed to a SAT solver via its API [5, 6]. At each test step, the formula is either extended by another clause, or the solver is called on the generated formula. The adapter takes over the role of translating test actions into the creation and use of data structures.

2.2 API-less/networked systems

Networked systems may provide a library through which a client communicates with a server. If this is the case, model-based testing can directly access that API [3]. To test services that rely solely on a given protocol (such as HTTP) but which cannot be called directly as a function, an adapter is needed. The adapter uses the protocol of the SUT internally, but provides an interface to this protocol so test actions represent protocol usage as calls to the adapter. An adapter may provide its own code to implement a protocol [4] or use another tool to interface with the SUT, as in offline testing [6].

2.3 GUI testing

In system testing, it is common to interact with the SUT via its graphical user interface (GUI). Test automation requires an adapter that provides access to the structure and elements of the GUI as well as the ability to send events (e. g., mouse movements and clicks, touch events) to the SUT. The implementation of an adapter providing these capabilities is usually specific to a particular GUI technology (e. g., JavaFX, Microsoft WinForms), and often made available in form of a technology specific "robot" library. Adapters on top of these libraries can support specific interactions and facilitate test case generation and model-based testing [10].

2.4 Model-based simulation

The behavior of the SUT may be affected by the physical environment outside the software. These environments change across the time unexpectedly, due to factors such as network delay or sensor failures. Simulations can test SUTs under such conditions. *Model-based simulation* describes simulators of environments as models. In this approach, model actions trigger changes

in the environment by updating simulation parameters [13]. The adapter interfaces with the simulation environment and also sets up and shuts down the simulation.

3 Conclusion

Model-based test generators often have to be adapted to concrete software execution environments. This adaptation is often time-consuming to implement. However, the same concept can be taken further by adapting test actions to data structure creation, interfaces with API-less systems, GUIs, or even a simulation environment. An adapter can represent a system interface or protocol at the right level of abstraction for the test tool. This enables the use of test case generation tools in new ways, and in settings that are not an obvious target for them.

References

[1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Li, and H. Zhu. An orchestrated survey of methodologies for automated software test case generation. *J Syst Softw.*

[2] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto. Modbat: A model-based API tester for event-driven systems. In *Proc. 9th Haifa Verification Conf.*, volume 8244 of *LNCS*, pages 112–128, Haifa, Israel, 2013. Springer.

[3] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, and M. Yamamoto. Model-based API testing of Apache ZooKeeper. In *10th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST 2017)*, Tokyo, Japan, 2017. IEEE.

[4] C. Artho, M. Hagiya, R. Potter, Y. Tanabe, F. Weitl, and M. Yamamoto. Software model checking for distributed systems with selector-based, non-blocking communication. In *Proc. 28th Int. Conf. on Automated Software Engineering*, ASE, pages 169–179, Palo Alto, USA, 2013. IEEE.

[5] C. Artho, M. Seidl, Q. Gros, E. Choi, T. Kitamura, A. Mori, R. Ramler, and Y. Yamagata. Model-based testing of stateful APIs with Modbat. In *Proc. 30th Int. Conf. on Automated Software Engineering*, ASE, pages 858–863, Lincoln, USA, 2015. IEEE.

[6] A. Biere, M. Seidl, and C. Artho. Model-based testing for verification backends. In *Proc. 7th Int. Conf. on Tests & Proofs (TAP 2013)*, volume 7942 of *LNCS*, pages 39–55. Springer, 2013.

[7] R. Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley, 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, USA, 1995.

[9] C. Pacheco and M. Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications Companion*, OOPSLA, pages 815–816, Montreal, Canada, 2007. ACM.

[10] R. Ramler, G. Buchgeher, and C. Klammer. Adapting automated test generation to GUI testing of industry applications. *Information & Software Technology*, 93:248–263, 2018.

[11] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, USA, 2006.

[12] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[13] J. Yoneyama. Model-based testing simulating unstable networks and devices for IoT software. Master's thesis, The University of Tokyo, Tokyo, Japan, 2018.

Towards Efficient Bit-Vector Interpolation

Peter Backeman, Philipp Rümmer, and Aleksandar Zeljić

Uppsala University, Sweden

Abstract

Reasoning with bit-vector (BV) arithmetic is an important problem in verification. Successful techniques for unbounded arithmetic, e.g., Craig interpolation, have turned out to be hard to generalise to machine arithmetic. We present a new approach to BV interpolation that works by lazy translation of bit-vector constraints to unbounded arithmetic. The present extended abstract is a shortened version of a upcoming conference publication [1].

1 Introduction

The inference of program invariants over machine arithmetic, or bit-vector (BV) arithmetic, is an important problem in verification, with Craig interpolation being one of the commonly used techniques. Over the last 15 years, efficient interpolation techniques have been developed for a variety of logics and theories, including propositional logic and linear real arithmetic [6], and Presburger arithmetic [2]. BV arithmetic has turned out notoriously difficult to handle in Craig interpolation. Decision procedures for BV are predominantly based on bit-blasting, resulting in that extracted interpolants stay on the level of propositional logic and are difficult to map back to compact high-level BV constraints. An alternative interpolation approach translates BV constraints to unbounded integer arithmetic formulas [5], but is limited to linear constraints and tends to produce integer formulas that are hard to solve and interpolate.

We introduce a new Craig interpolation method for BV arithmetic, focusing on arithmetic BV operations including addition, multiplication, and division. Like [5], we compute interpolants by reducing BVs to unbounded integers; unlike earlier approaches, we define a calculus that carries out this reduction lazily. By initially representing BV operations as uninterpreted predicates, which are replaced by Presburger arithmetic expressions on demand, our approach can dynamically choose between multiple possible encodings. The calculus also includes native rules for non-linear constraints and BV equations, so that formulas can often be proven without having to resort to a full encoding as integer constraints. Our approach gives rise to both Craig interpolation and quantifier elimination (QE) methods for BV constraints, with both procedures displaying competitive performance in our experiments.

This extended abstract summarizes previous work and introduces future directions. For a more detailed description we refer the reader to the paper to be presented at FMCAD 2018 [1].

2 Translations

Our base calculus is a sequent calculus for classical first-order logic (with quantifiers) combined with equalities and inequalities of Presburger arithmetic (PA). BV formulas are translated eagerly to the combination of a BV *core language* and non-linear integer arithmetic (NIA).

The NIA fragment consists of PA extended by a ternary multiplication predicate $\times$, with the semantics that the third argument represents the result of multiplying the first two arguments, i.e., $\times(s,t,r) \Leftrightarrow s \cdot t = r$. The BV core language is PA extended with a family of binary predicates $P_{bv} = \{bmod_a^b \mid a, b \in \mathbb{Z}, a < b\}$. The semantics of $bmod_a^b$ is to relate any $x \in \mathbb{Z}$ to its remainder modulo $b-a$ in $\{a, \ldots, b-1\}$: $bmod_a^b(s,r) \Leftrightarrow a \leq r < b \ \wedge r \equiv s \pmod{b-a}$.

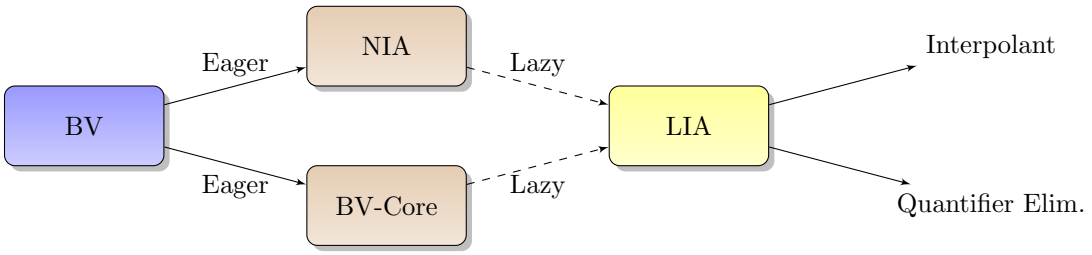


Figure 1: Translation of BV Formulas

The main idea of our approach is to *lazily* translate constraints from the NIA fragment or from the BV core language to Presburger arithmetic, thereby enabling calculus rules to dynamically choose between multiple possible encodings of the BV operations. For instance, occurrences of $bmod_a^b$ can either be eliminated by enumerating possible overflow cases, replaced by a simple equality when only one case is possible, or translated to the full PA encoding of a remainder operation. Afterwards, previous techniques for Craig Interpolation [2] and Quantifier Elimination (for Presburger arithmetic) can be applied. The whole scheme is shown in Fig 1.

We also introduce new calculus rules for deriving linear constraints from NIA formulas, for example by deriving implied equalities with Gröbner Bases. The rules are necessarily incomplete for proving that an NIA formula is valid, but complete for finding satisfying assignments.

Example 1. *We consider one of the examples from [5], the interpolation problem $A \wedge B$:*

$$A = \neg \mathsf{bvule}_8(\mathsf{bvadd}_8(y_4, 1), y_3) \wedge y_2 = \mathsf{bvadd}_8(y_4, 1)$$
$$B = \mathsf{bvule}_8(\mathsf{bvadd}_8(y_2, 1), y_3) \wedge y_7 = 3 \wedge y_7 = \mathsf{bvadd}_8(y_2, 1)$$

An eager encoding into LIA would typically add variables to handle wrap-around semantics, e.g., mapping $y_4' = \mathsf{bvadd}_8(y_4, 1)$ to $y_4' = y_4 + b1 - 2^8\sigma_1 \wedge 0 \leq y_4' < 2^8 \wedge 0 \leq \sigma_1 \leq 1$. Additional variables tend to be hard for interpolation, and the LIA interpolant presented in [5] is the formula $I_{LIA} = -255 \leq y_2 - y_3 + 256\lfloor -1\frac{y_2}{256}\rfloor$; the formula can be mapped back to a pure BV formula. Our calculus finds the simpler interpolant $I'_{LIA} = y_3 < y_2$ for this problem.

To prove the unsatisfiability of $A \wedge B$, translation to our BV core language gives:

$$A_{core} = \quad \psi_A \wedge bmod_0^{2^w}(y_4 + 1, c_1) \wedge c_1 > y_3 \wedge y_2 = c_1$$
$$B_{core} = \quad \psi_B \wedge bmod_0^{2^w}(y_2 + 1, c_2) \wedge c_2 \leq y_3 \wedge y_7 = 3 \wedge y_7 = c_2$$

where $\psi_A = in_8(y_2) \wedge in_8(y_3) \wedge in_8(y_4) \wedge in_8(c_1)$ and $\psi_B = in_8(y_2) \wedge in_8(y_3) \wedge in_8(y_7) \wedge in_8(c_2)$ are domain constraints, and $in_w(x) =_{def} (0 \leq x < 2^w)$. Unsatisfiability of $A_{core} \wedge B_{core}$ can be proven by splitting over possible cases of $bmod_0^{2^w}$:

$$\frac{\ldots, 0 \leq c_2 < 256, y_2 + 1 = c_2 \vdash \qquad \ldots, 0 \leq c_2 < 256, y_2 + 1 = c_2 + 256 \vdash}{\ldots, bmod_0^{2^w}(y_2 + 1, c_2) \vdash} \text{ BMOD-SPLIT}$$

Due to $y_7 = 3 \wedge y_7 = c_2$, the cases reduce to $y_2 = 2$ and $y_2 = 258$, thus contradicting A_{core}, B_{core}. Once a closed proof has been found, the interpolant I'_{LIA} can be extracted using rules from [2].

3 Experiments

Verification of C Programs We present results of running the ELDARICA model checker[1] on a benchmark set of 551 C programs, assuming 32 bit wrap-around integer semantics (ilp32), and using the implementation of our calculus in PRINCESS[2] as interpolation procedure. The benchmarks are the programs used in [4] for evaluating different predicate generation strategies.

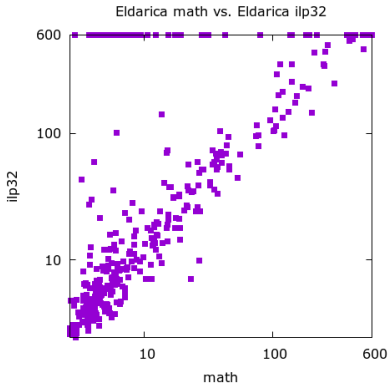


Figure 2: Runtime of math and ilp32 semantics

As a comparison, we also verified the programs assuming mathematical integer semantics, i.e., with unbounded integers and no overflows. The experiments showed that our interpolation approach for BVs can solve almost as many programs in 32 bit semantics as with mathematical semantics, with a similar number of CEGAR iterations, with interpolants of comparable size, and with only a small runtime overhead (Fig. 2, times in seconds).

4 Towards Interpolation for Full Bit-Vector Logic

Up to this point, BV functions such as extract and concat are not handled in our calculus, and neither are bit-wise operations like bvand or bvxor. We plan to extend our method to such functions by extending the core language with a further set $P'_{bv} = \{extract_{[l:r]} \mid l, r \in \mathbb{N}, l \geq r\}$ of binary predicates expressing the extraction of $l - r + 1$ bits from a bit-vector:

$$extract_{[l:r]}(s,t) \quad \Leftrightarrow \quad 0 \leq t < 2^{l-r+1} \wedge \left(\exists y, z \in \mathbb{N}.\ 0 \leq z < 2^r \wedge s = 2^{l+1}y + 2^r t + z\right)$$

Similarly to $bmod_a^b$, the $extract_{[l:u]}$ predicates can lazily be handled with additional calculus rules, rephrasing procedures for the polynomial core BV fragment [3]. Combined with our interpolating calculus for PA and uninterpreted predicates [2], such rules yield an interpolation procedure for the core fragment. Bit-wise operations can be encoded eagerly using $extract_{[l:u]}$.

References

[1] Peter Backeman, Philipp Rümmer, and Aleksandar Zeljic. Bit-vector interpolation and quantifier elimination by lazy reduction. In *FMCAD*, 2018. To appear.

[2] Angelo Brillout, Daniel Kroening, Philipp Rümmer, and Thomas Wahl. Beyond quantifier-free interpolation in extensions of Presburger arithmetic. In *VMCAI*, LNCS. Springer, 2011.

[3] Roberto Bruttomesso and Natasha Sharygina. A scalable decision procedure for fixed-width bit-vectors. In *2009 International Conference on Computer-Aided Design, ICCAD 2009*.

[4] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In *NFM*, volume 10227 of *LNCS*, pages 265–281, 2017.

[5] Alberto Griggio. Effective word-level interpolation for software verification. In Per Bjesse and Anna Slobodová, editors, *FMCAD*, pages 28–36. FMCAD Inc., 2011.

[6] Kenneth L. McMillan. An interpolating theorem prover. *Theor. Comput. Sci.*, 345(1), 2005.

[1] `https://github.com/uuverifiers/eldarica`

[2] `http://www.philipp.ruemmer.org/princess.shtml`

Higher-Dimensional Timed Automata Extended Abstract

Uli Fahrenberg

École Polytechnique, Palaiseau, France*

We introduce a new formalism of higher-dimensional timed automata, based on van Glabbeek's higher-dimensional automata and Alur's timed automata. We prove that their reachability is PSPACE-complete and can be decided using zone-based algorithms. We also show how to use tensor products to combat state-space explosion and how to extend the setting to higher-dimensional hybrid automata.

In approaches to non-interleaving concurrency, more than one event may happen concurrently. There is a plethora of formalisms for modeling and analyzing such concurrent systems, *e.g.,* Petri nets [16], event structures [15], configuration structures [23], or more recent variations such as dynamic event structures [5] and Unravel nets [7]. They all share the convention of differentiating between concurrent and interleaving executions; using CCS notation [14], $a|b \neq a.b + b.a$.

For modeling and analyzing embedded or cyber-physical systems, formalisms which use real time are available. These include timed automata [4], time Petri nets [13], timed-arc Petri nets [12], or various classes of hybrid automata [2]. Common for them all is that they identify concurrent and interleaving executions; here, $a|b = a.b + b.a$.

We are interested in formalisms for real-time non-interleaving concurrency. Hence we would like to differentiate between concurrent and interleaving executions and be able to model and analyze real-time properties. Few such formalisms seem to be available in the literature. (The situation is perhaps best epitomized by the fact that there is a natural non-interleaving semantics for Petri nets [11] which is also used in practice [8,9], but almost all work on real-time extensions of Petri nets [12,13,19,21] use an interleaving semantics.)

We introduce higher-dimensional timed automata (HDTA), a formalism based on the (non-interleaving) higher-dimensional automata of [22] and [17] and the timed automata of [3,4]. We show that HDTA can model interesting phenomena which cannot be captured by neither of the formalisms on which they are based, but that their analysis remains just as accessible as the one of timed automata. That is, reachability for HDTA is PSPACE-complete and can be decided using zone-based algorithms.

In the above-mentioned interleaving real-time formalisms, continuous flows and discrete actions are orthogonal in the sense that executions alternate between real-time delays and discrete actions which are immediate, *i.e.,* take no time. (In the hybrid setting, these are usually called flows and mode changes, respectively.) Already [20] notice that this significantly simplifies the semantics of such systems and hints that this is a main reason for the success of these formalisms (see the more recent [21] for a similar statement).

In the (untimed) non-interleaving setting, on the other hand, events have a (logical, otherwise unspecified) duration. This can be seen, for example, in the ST-traces of [22] where actions have a start (a^+) and a termination (a^-) and are (implicitly) running between their start and termination, or in the representation of concurrent systems as Chu spaces over $3 = \{0, \frac{1}{2}, 1\}$, where 0 is interpreted as "before", $\frac{1}{2}$ as "during", and 1 as "after", see [18]. Intuitively, only if events have duration can one make statements such as "while a is running, b starts, and then while b is running, a terminates".

*Supported by the *Chaire ISC : Engineering Complex Systems* – École polytechnique – Thales – FX – DGA – Dassault Aviation – DCNS Research – ENSTA ParisTech – Télécom ParisTech

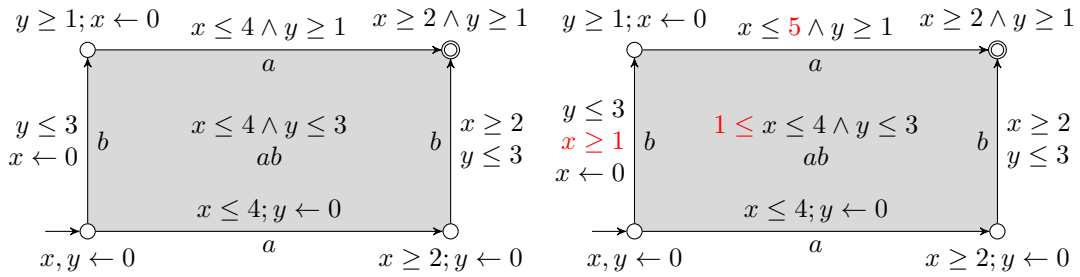


Figure 1: Two two-dimensional timed automata

In our non-interleaving real-time setting, we hence abandon the assumption that actions are immediate. Instead, we take the view that actions start and then run during some *specific* time before terminating. While this runs counter to the standard assumption in most of real-time and hybrid modeling, a similar view can be found, for example, in Cardelli's [6].[1]

Given that we abandon the orthogonality between continuous flows and discrete actions, we find it remarkable to see that the standard techniques used for timed automata transfer to our non-interleaving setting. Equally remarkable is, perhaps, the fact that even though *"[t]he timed-automata model is at the very border of decidability, in the sense that even small additions to the formalism [...] will soon lead to the undecidability of reachability questions"* [1], our extension to higher dimensions and non-interleaving concurrency is completely free of such trouble.

Figure 1 gives a few examples of two-dimensional timed automata. The first models two actions, a and b, which can be performed concurrently. It consists of four states (0-cubes), four transitions (1-cubes), and one ab-labeled square (2-cube). This HDTA models that performing a takes between two and four time units, whereas performing b takes between one and three time units. To this end, we use two clocks x and y which are reset when the respective actions are started and then keep track of how long they are running.

In the second HDTA of Fig. 1 (where we show changes to the first in red), invariants have been modified so that b can only start after a has been running for one time unit, and if b finishes before a, then a may run one time unit longer. Hence an invariant $x \geq 1$ is added to the two b-labeled transitions and to the ab-square (at the right-most b-transition $x \geq 1$ is already implied), and the condition on x at the top a-transition is changed to $x \leq 5$. Note that the left edge is now permanently disabled: before entering it, x is reset to zero, but its edge invariant is $x \geq 1$. This is as expected, as b should not be able to start before a.

This abstract is based on work which has been presented at ADHS 2018 [10]. Compared to this paper, we now have a much more precise idea of the translation from HDTA to standard timed automata and of higher-dimensional timed *languages*.

References

[1] Luca Aceto, Anna Ingólfsdóttir, Kim G. Larsen, and Jiří Srba. *Reactive Systems*. Cambridge Univ. Press, 2007.

[1] The author wishes to thank Kim G. Larsen for pointing him towards this paper.

[2] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995.

[3] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *ICALP*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[5] Youssef Arbach, David Karcher, Kirstin Peters, and Uwe Nestmann. Dynamic causality in event structures. In *FORTE*, volume 9039 of *LNCS*, pages 83–97. Springer, 2015.

[6] Luca Cardelli. Real time agents. In *ICALP*, volume 140 of *LNCS*, pages 94–106. Springer, 1982.

[7] Giovanni Casu and G. Michele Pinna. Petri nets and dynamic causality for service-oriented computations. In *ACM SAC*. ACM, 2017.

[8] Javier Esparza. A false history of true concurrency: From Petri to tools (invited talk). In *SPIN*, volume 6349 of *LNCS*, pages 180–186. Springer, 2010.

[9] Javier Esparza and Keijo Heljanko. *Unfoldings – A Partial-Order Approach to Model Checking*. Monographs Theor. Comput. Sci. Springer, 2008.

[10] Uli Fahrenberg. Higher-dimensional timed automata. *CoRR*, abs/1802.07038, 2018.

[11] Ursula Goltz and Wolfgang Reisig. The non-sequential behavior of Petri nets. *Inf. Cont.*, 57(2/3):125–147, 1983.

[12] Hans-Michael Hanisch. Analysis of place/transition nets with timed arcs and its application to batch process control. In *ATPN*, volume 691 of *LNCS*, pages 282–299. Springer, 1993.

[13] Philip M. Merlin and David J. Farber. Recoverability of communication protocols–implications of a theoretical study. *IEEE Trans. Comm.*, 24(9):1036–1043, 1976.

[14] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[15] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theor. Comput. Sci.*, 13:85–108, 1981.

[16] Carl A. Petri. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.

[17] Vaughan R. Pratt. Modeling concurrency with geometry. In *POPL*, pages 311–322. ACM Press, 1991.

[18] Vaughan R. Pratt. Higher dimensional automata revisited. *Math. Struct. Comput. Sci.*, 10(4):525–548, 2000.

[19] Joseph Sifakis. Use of Petri nets for performance evaluation. In *Measuring, Modelling and Evaluating Computer Systems*, pages 75–93. North-Holland, 1977.

[20] Joseph Sifakis and Sergio Yovine. Compositional specification of timed systems. In *STACS*, volume 1046 of *LNCS*, pages 347–359. Springer, 1996.

[21] Jiří Srba. Comparing the expressiveness of timed automata and timed extensions of Petri nets. In *FORMATS*, volume 5215 of *LNCS*, pages 15–32. Springer, 2008.

[22] Rob J. van Glabbeek. On the expressiveness of higher dimensional automata. *Theor. Comput. Sci.*, 356(3):265–290, 2006.

[23] Rob J. van Glabbeek and Gordon D. Plotkin. Configuration structures, event structures and Petri nets. *Theor. Comput. Sci.*, 410(41):4111–4159, 2009.

Using Coloured Petri Nets for Resource Analysis of Active Objects

Anastasia Gkolfi[1], Einar Broch Johnsen[1], Lars Michael Kristensen[2], and Ingrid Chieh Yu[1]

[1] Department of Informatics, University of Oslo, Norway
{natasa, einarj, ingridcy}@ifi.uio.no
[2] Western Norway University of Applied Sciences, Norway
lmkr@hvl.no

1 Introduction

Pay-on-demand resource provisioning is an important driver for cloud computing. Virtualized resources in cloud computing open for resource awareness, such that applications may contain resource management strategies to modify their deployment and reduce resource consumption. The ABS language supports the modelling of deployment decisions and resource management for active objects. In this paper, we present a CPN model [3, 4] of the deployment fragment of ABS [5]. A key characteristics of our approach is that the compact modelling supported by CPNs allowed us to develop a CPN model capable of simulating any ABS program by only changing the initial marking. The main benefit of our approach is the ability to use model checking techniques to identify starvation of resource aware active objects, and to synthesise reconfiguration sequences that eliminates starvation and which in turn can be used to automatically obtain load-balancer implementations.

2 The CPN model

In [1] the authors presented a CPN, modelling the concurrency of ABS. In the current work, we present a new CPN, taking as input tokens that can be produced from the model introduced in [1] (the imperative part of ABS). As it was the case in [1], active objects are represented as tokens and we add information concerning the cost of each process and the deployment component where active objects are located. This information, together with the deployment semantics of ABS can be used to verify starvation freedom of active objects and explore resource management strategies.

In the rest of the paper, we use a running example inspired from cellphone clients behaviour in order to illustrate the relation between the CPN model and ABS programs and to show how we can use the model checker of CPN Tools for load balancing scenarios.

The average demand on phone calls and SMS messages from cellphone clients during the year is relatively low and the available resources suffice in a current distribution. There are some particular moments of the year like, for example, around the midnight of new year's eve, where this behaviour changes and a large number of SMS is requested by the clients while the call requests are negligible. Then, the initial distribution is not adequate, since there is a lack of resources for the SMS and an overplus for the calls.

The above scenario has been implemented in ABS [5]: telephone and SMS servers have been realised with the two corresponding classes and the operational cost annotated at the beginning of the statements. Cellphone clients have been implemented with corresponding classes allowing objects to make method calls to the SMS and telephone services.

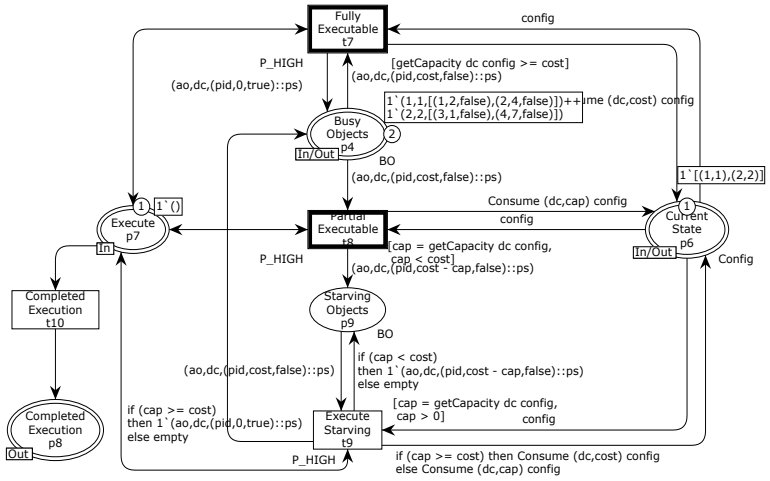


Figure 1: CPN module for process execution

Details on the implementation of the CPN model as well as the abstraction relation between ABS program configurations and CPN markings and a full proof supporting the soundness of the model can be found in a technical report [2]. Here, in Fig. 1, we provide a part of the model representing the resource consumption caused by the process execution. As we mentioned above, tokens are tuples representing active objects and they contain the object id, the deployment component where the object is located and a list of the processes together with the corresponding costs. These "object-tokens" can be located either to the place Busy Objects or to the place Starving Objects. Place Current State keeps track of the available resource distribution. Based on that, transitions Fully Executable or Partially Executable can fire depending on whether the available resources suffice for the full execution of a process or not, moving the corresponding objects to the appropriate place (resp. Busy Objects or Starving Objects place). We can use this property of the model at the state space exploration for detecting starvation freedom or ask for resource distribution which lead to starvation free states, as we will see in the next section.

3 Resource Analysis and Load Balancing

The first important information related to resource management is whether the current resource distribution provides sufficient resources for the full execution of the processes the objects have in their process pools. In other words, we need to check for *starvation freedom*. By model construction, place Starving Objects keeps track of the starving objects, if any. For the analysis, we implement Standard ML queries in CPN Tools. For our new year's eve midnight telephone and SMS services example, CPN Tools returns a non-empty list containing several states which imply that starvation is possible, so an interesting question is *whether there exists a resource reallocation strategy leading to starvation freedom*. For that, we need to check for the existence of any terminal SCC containing states where there are no starving objects and then ask for a path leading to that state. Such a path contains the following resource transfers:

[Component Reconfiguration‘Reconfigure
$(1, \{b = true, cap = 1, \text{config} = [(1,1),(2,2)], \text{fromdc} = 1, \text{todc} = 2\})$

$(1, \{b = true, cap = 3, \text{config} = [(1,0),(2,3)], \text{fromdc} = 2, \text{todc} = 1\})$
$(1, \{b = true, cap = 2, \text{config} = [(1,3),(2,0)], \text{fromdc} = 1, \text{todc} = 2\})]$

where, variables *fromdc* and *todc* indicate the source and the target deployment component of each transfer and *cap* the amount of the resources we need to move.

```
1  class Balancer(DC telcomp, DC smscomp) {
2    Unit run() {
3      telcomp!transfer(smscomp,1);
4      await duration(2,2);
5      smscomp!transfer(telcomp, 3);
6      await duration(2,2);
7      telcomp!transfer(smscomp,2);}}
8  {// Main block
9    ... // deployment components, etc. as before
10   new Balancer(telcomp,smscomp);}
```

Figure 2: Implementation of Load Balancer.

From the above path information we can implement very easily a load balancer like the one of Fig. 2. We assume the telephone service to be located to the deployment component "telcomp" (in the model represented with integer "1") and the SMS service to the deployment component "smscomp" having as an identifier in the model the integer "2". Lines 3, 5, and 7 in Fig. 2 correspond to the relevant values of the variables *fromdc* and *todc* from the above path.

Our present work extends [1] by taking as input the communication status of resource aware active objects and performing resource analysis. We demonstrated how to statically construct a load balancer. A direction for future work will be to extend the model to support dynamic load balancing and investigate optimal vertical scaling using the CPN model checker. Another direction will be to perform a comprehensive experimental evaluation on a larger set of ABS programs.

References

[1] Anastasia Gkolfi, Crystal Chang Din, Einar Broch Johnsen, Martin Steffen, and Ingrid Chieh Yu. Translating active objects into Colored Petri Nets for communication analysis. In *Proc. of FSEN'17*, volume 10522 of *LNCS*, pages 84–99. Springer, 2017.

[2] Anastasia Gkolfi, Einar Broch Johnsen, Lars Michael Kristensen, and Ingrid Chieh Yu. Using Coloured Petri Nets for resource analysis of active objects (full version). Technical Report 484, Dept. of informatics, University of Oslo, 2018.

[3] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volume 1.* Springer, 2010.

[4] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems.* Springer, 2009.

[5] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. Integrating deployment architectures and resource consumption in timed object-oriented models. *Journal of Logical and Algebraic Methods in Programming*, 84(1):67—91, 2015.

Formalizing and Analyzing Security Ceremonies with Heterogeneous Devices in ANP and PDL

Antonio Gonzalez Burgueño and Peter Csaba Ölveczky

University of Oslo, Norway

1 Introduction

Most security breaches nowadays occur not by breaking cryptographic protocols or through buffer overflow, but through various forms of "social engineering," such as phishing emails, malicious apps and web sites, and so on. Furthermore, web applications typically interact with human users. To reason about security in modern systems, we must therefore include the humans as key parts of the security process, which requires defining new models of such processes. For example, in standard crypto-protocol formalisms, the behavior of each node/actor is typically given as a *sequence* of actions. This is not sufficient if we include humans in the system, since humans may exhibit *nondeterministic* behaviors (does the user click on the link, or not? does the user perform an action in the wrong way?).

Security ceremonies [3] extend cryptographic protocols with models of human users. *Actor-network procedures* (ANPs), introduced by Pavlovic and Meadows, are one of the more popular ways of formalizing security ceremonies (see, e.g., [1, 2, 4, 5, 7, 8]), and *procedure derivation logic* (PDL) [4, 8] allows us to reason logically about ANPs. In essence, ANPs define the possible behaviors as partial orders over events, and PDL formulas allow nodes to assert the order of events in a protocol run (e.g., "if n received m, then some node must previously have sent m").

A security ceremony typically includes different kinds of nodes, such as computers, humans, and authentication devices like smart cards, random generation numbers, biometric devices, and so on. One key thing is that different actors may have very different capabilities: a computer can encrypt and decrypt messages whereas humans cannot; a biometric device can capture biometric information, whereas a random number generator used in e-banking cannot; and so on. In addition, a security ceremony may include many forms of communication: the visual channel between a user and her computer screen is authenticated (but not necessarily secret, since someone else may also look at my screen), whereas the channel between a user's computer and her e-bank is both authenticated and secret, assuming appropriate use of cryptography.

ANPs are a fairly simple and general formalism, and does not support specifying that different nodes have different capabilities. We therefore define *extended ANPs* (E-ANPs), which extend ANPs by allowing the user to specify the capabilities of the different nodes, and add to ANP's send and receive events explicit events for: (a) learning things from previously received messages, and (b) creating new knowledge from existing knowledge and the node's capabilities.

PDL supports reasoning about the temporal order of *events*, and does not allow us to reason about the *knowledge* of the nodes at certain times; for example, a node that has the capability to decrypt an encrypted message can only do so if it currently knows the decryption key for the message. To reason about possible behaviors of security ceremonies more accurately, we should be able to keep track of the knowledge of each node throughout the run of the ceremony. We therefore change PDL to allow reasoning about E-ANPs as follows: (i) the atoms in our new logic E-PDL are no longer events, but are instead pairs $\langle \mathit{event}, \mathit{knowledge} \rangle$ of an event and the global knowledge at the time point when the event takes place; (ii) a new set of axioms axiomatize the properties of E-ANPs, including what is being learnt.

2 E-ANP and E-PDL

The static part of an *actor-network procedure* (ANP) [6, 8] is defined as an *actor-network*, which consists of a (possibly hierarchical) network of nodes and channels between them. An *actor-network procedure* extends an actor-network by adding a *process*, which defines the behaviors of each subsystem. ANPs assume an algebraic theory (Σ, E) of parametric operations, such as encryption, decryption, creating a nonce, etc. An *event* or *action* has the form $a[t]$, where a is an event identifier and t its parameter. Important event identifiers are *send* and *receive*.

The (local) behaviors of a system are given by a *process* ρ, which is defined as a partially ordered multiset of *localized events*. Formally, a process $\mathcal{F}$ is a pair $\mathcal{F} = \langle \mathcal{F}_{\mathbb{E}}, \mathcal{F}_{\mathcal{P}} \rangle : \mathbb{F} \to \mathbb{E} \times \mathcal{P}$, where $(\mathbb{F}, \to)$ is a well-founded partial order, representing the structure time, $\mathbb{E}$ is the set of events, and $\mathcal{P}$ are the nodes/subsystems. Informally, $e_P \to e'_Q$ if the event e must happen at P before e' happens at Q; that is, there are two time points f_1 and f_2 where $f_1 \to f_2$, $\mathcal{F}_{\mathbb{E}}(f_1) = e$, $\mathcal{F}_{\mathbb{E}}(f_2) = e'$, $\mathcal{F}_{\mathcal{P}}(f_1) = P$, and $\mathcal{F}_{\mathcal{P}}(f_2) = Q$. A *run* can be seen as a partially ordered multiset of localized events that extends the internal synchronizations to the whole network. For notational convenience, it is often assumed that an event takes place at most once.

We define an *extended ANP* (E-ANP) to be a pair $(\mathcal{A}, \mathbb{C})$ where:

- $\mathcal{A}$ is an ANP such that the different capabilities of the devices and their algebraic properties are included in its underlying algebraic theory (Σ, E), and where *apply_to_to learn_* and *apply_to_* are event identifiers (for learning events), and
- $\mathbb{C}$ is a *capability distribution* $\mathbb{C} : \mathcal{N} \to \mathcal{P}(\Sigma)$ assigning to each node n in $\mathcal{A}$ its capabilities.

The *extended procedure derivation logic* (E-PDL) extends and modifies PDL to reason not only about the temporal order of events, but also of the knowledge of the participants at each time point. A *knowledge assignment* $\mathbb{KA}$ is a function $\mathbb{KA} : \mathcal{N} \to \mathcal{P}(\mathcal{T}_\Sigma)$ that assigns to each node n the set of terms known by n.

Assuming that an event takes place at at most one time point, the E-PDL formulas are first-order logic formulas with atomic propositions including pairs $\langle e_1, ka_1 \rangle$ and $\langle e_1, ka_1 \rangle \to \langle e_2, ka_2 \rangle$—denoting that the event e_i took place at some time point f_i and that the knowledge assignment at *the end of* time point f_i was ka_i; the latter also denotes as usual that e_1 preceded e_2. The axioms of E-PDL are shown in Table 1 can be summarized as follows:

- Equality: If a node p knows t, it also knows all E-equivalent terms t'.
- Send: If a node p sends a term t, then t is part of its knowledge ($t \in \mathbb{KA}(p)$) at the end of the current point in time. Furthermore, nothing is learnt in the entire system during this time point (*nothingLearnt*(...)).
- Receive: If a node p receives t, then t is included in its knowledge ($t \in \mathbb{KA}_1(p)$) at the end of the current time point. Furthermore, t was sent by some node X at some earlier time point. In addition, the only change in knowledge caused by the receive event is that p has learnt the term t (and all the *E-equivalent* terms) (*onlyLearnt*(...)).
- Learn: If node p learns a term t (which is the only item added to the global knowledge) from a term u using an operation O and the terms $u_1, \ldots, u_n$, i.e., $O(u, u_1, \ldots, u_n) =_E x$, then the terms $\{u, u_1, \ldots, u_n\} \subseteq \mathbb{KA}(p)$ are part of p's knowledge at the time in which the event takes place, and the operation O is part of p's capabilities.
- Creation: If a node p creates a term $O(t_1, \ldots, t_n)$, then p knows this term at the end of the current point in time (and $O(t_1, \ldots, t_n)$ is the only new thing learnt). Furthermore, the node p has the capability to perform the operation O ($O \in \mathbb{C}(p)$), and knows $t_1, \ldots, t_n$.
- Knowl.Preservation: If a node p knows t, it will also know t at all later time points.

Equality	$\forall t, t', p \,.\, t \in \mathbb{KA}(p) \ \wedge \ t =_E t' \implies t' \in \mathbb{KA}(p)$
Send	$\langle send(z)_p, \mathbb{KA}\rangle \implies z \in \mathbb{KA}(p) \wedge nothingLearnt(\langle send(z)_p, \mathbb{KA}\rangle)$
Receive	$\langle receive(z)_p, \mathbb{KA}_1\rangle \implies z \in \mathbb{KA}_1(p) \wedge \ onlyLearnt(\langle receive(z)_p, \mathbb{KA}\rangle, z, p)$ $\wedge (\exists q, \mathbb{KA}_2.\ \langle send(z)_q, \mathbb{KA}_2\rangle \rightarrow \langle receive(z)_p, \mathbb{KA}_1\rangle)$
Learn	$\langle (apply\ O\ to\ u\ toLearn\ t)_p, \mathbb{KA}\rangle$ $\implies t \in \mathbb{KA}(p) \ \wedge \ O \in \mathbb{C}(p)$ $\wedge (\exists\ u_1, \ldots, u_n.\ \{u_1, \ldots, n_n\} \subseteq \mathbb{KA}(p) \wedge \ O(u, u_1, \ldots, u_n) =_E t)$ $\wedge\ onlyLearnt(\langle (apply\ O\ to\ u\ toLearn\ t)_p, \mathbb{KA}\rangle, t, p)$
Creation	$\langle (applyOp\ O\ to\ t_1, \ldots, t_n)_p, \mathbb{KA}\rangle$ $\implies O(t_1, \ldots, t_n) \in \mathbb{KA}(p) \ \wedge \ O \in \mathbb{C}(p) \wedge \{t_1, \ldots, t_n\} \subseteq \mathbb{KA}(p)$ $\wedge\ onlyLearnt(\langle (applyOp\ O\ to\ t_1, \ldots, t_n)_p, \mathbb{KA}\rangle, O(t_1, \ldots, t_n), p)$
Knowl.Preservation	$\langle e_1, \mathbb{KA}_1\rangle \rightarrow \langle e_2, \mathbb{KA}_2\rangle \implies \mathbb{KA}_1(p) \subseteq \mathbb{KA}_2(p)$

Table 1: E-PDL Axioms

The formula $onlyLearnt(\langle e, \mathbb{KA}\rangle, t, p)$ denotes that the only knowledge added to the system as a resulting of the event e being performed is that p learnt t (and all E-equivalent terms) can be defined as follows. Basically, if q knows u then either $p = q$ and $u =_E t$, or q knew u when an *earlier* event e_2 took place, or q knew u initially ($u \in \mathbb{KA}_0(q)$):

$$onlyLearnt(\langle e, \mathbb{KA}\rangle, t, p) \ \triangleq \ \forall u, q.\ u \in \mathbb{KA}(q)$$
$$\implies (u =_E t \ \wedge \ p = q) \ \vee \ u \in \mathbb{KA}_0(q) \vee \ \exists e_2, \mathbb{KA}_2.\ \langle e_2, \mathbb{KA}_2\rangle \rightarrow \langle e, \mathbb{KA}\rangle \ \wedge \ u \in \mathbb{KA}_2(q).$$

The formula $nothingLearnt(\langle e, \mathbb{KA}\rangle)$ denotes that nothing was learnt by performing e:

$$nothingLearnt(\langle e, \mathbb{KA}\rangle) \ \triangleq$$
$$\forall t, q.\ t \in \mathbb{KA}(q) \implies t \in \mathbb{KA}_0(q) \ \vee \ (\exists e_2, \mathbb{KA}_2.\ \langle e_2, \mathbb{KA}_2\rangle \rightarrow \langle e, \mathbb{KA}\rangle \ \wedge \ t \in \mathbb{KA}_2(q)).$$

References

[1] Anlauff, M., Pavlovic, D., Waldinger, R., Westfold, S.: Proving authentication properties in the Protocol Derivation Assistant. In: Proc. FCS-ARSPA'06. ACM (2006)

[2] Cervesato, I., Meadows, C., Pavlovic, D.: An encapsulated authentication logic for reasoning about key distribution protocols. In: Proc. CSFW'05. IEEE (2005)

[3] Ellison, C.M.: Ceremony design and analysis. IACR Cryptology ePrint Archive 2007, 399 (2007)

[4] Meadows, C., Pavlovic, D.: Deriving, attacking and defending the GDOI protocol. In: Proc. ESORICS'04. LNCS, vol. 3193. Springer (2004)

[5] Meadows, C., Pavlovic, D.: Formalizing physical security procedures. In: Proc. STM'12. LNCS, vol. 7783. Springer (2012)

[6] Pavlovic, D., Meadows, C.A.: Actor-network procedures (extended abstract). In: Proc. ICDCIT'12. Springer LNCS, vol. 7154. Springer (2012)

[7] Pavlovic, D., Meadows, C.: Deriving secrecy properties in key establishment protocols. In: Proc. ESORICS'06. LNCS, vol. 4189. Springer (2006)

[8] Pavlovic, D., Meadows, C.: Actor-network procedures: Modeling multi-factor authentication, device pairing, social interactions. CoRR abs/1106.0706 (2011), http://arxiv.org/abs/1106.0706

Connecting Choreography Languages With Verified Stacks

Alejandro Gómez-Londoño[1] and Johannes Åman Pohjola[2]

[1] Chalmers University of Technology, Gothenburg, Sweden
alejandro.gomez@chalmers.se
[2] Data61/CSIRO, Sydney, Australia
johannes.amanpohjola@data61.csiro.au

Abstract

With ever increasing availability of verified stacks capable of guaranteeing end-to-end correctness on applications—like compilers (CakeML, CompCert) or even critical software systems (seL4)—one can now realistically write a program, along with a proof describing any desirable property, and have it compiled into a correct executable implementation of the original program. However, most of these approaches can only really deal with sequential programs and provide no support for reasoning about the correctness of multiple (concurrent) programs. To address these shortcomings, we propose a choreographic language where the behavior of a system consisting of several endpoints, is described on a global level, that can be subsequently projected and compiled into its individual components. We are developing an end-to-end proof of correctness that ensures i) the deadlock-freedom of the generated set of endpoints and ii) the preservation of all behavior of the system down to the binary level. Our implementation uses the verified CakeML compiler as a backend and takes advantages of its verified stack.

This extended abstract presents our ongoing work on connecting choreography languages with verified stacks. Our overarching goal is to develop an environment where programmers can write communicating systems as high-level protocol descriptions in the style of `Alice` → `Bob` notation, and then, easily generate executable code that is formally verified to correctly implement the protocol down to the machine-code level. To achieve our goal, we connect these two strands of work to provide both a convenient abstraction for representing such systems, and strong guarantees about their behaviour.

Choreographies present a global description of the interactions of a system in terms of the messages exchanged between its components [1] (endpoints). Its syntax resembles the description of a protocol, but it provides a more concrete definition of the system. For example, in lines 2 and 3 of Figure 1, `A` sends `B` the name of an `item` and `B` sends back its price by evaluating `price(x)`. This choreography, while simple, completely captures the interaction between `A` and `B`, abstracting away individual operations like `send` and `receive` as communications (→).

```
1  -- choreography
2  A[item]      → B.x
3  B[price(x)] → A.y
4
5  -- projection A
6  send(B,item)
7  receive(B,y)
8
9  -- projection B
10 receive(A,x)
11 send(B,price(x))
```

Figure 1: price-query choreography

Choreographies can be thought of as protocol descriptions, in the sense that they are descriptions that a set of programs—in the form of threads, processes, servers, etc—implement, and that as a whole, should exhibit the intended behaviour. Nevertheless, making sure a system complies with its specification is in general no easy task, and we have seen time and time again examples of protocols being implemented poorly [2]. What distinguishes choreographies is that they are programming languages, and hence provide concrete enough descriptions to generate implementations directly from them.

Endpoint projection (EPP) [6] generates an implementation for a choreography by translating each endpoint into a program, that when joined through parallel composition with all the others, should exhibit the same interactions as the original specification. This correctness property is know as the *endpoint projection theorem* (EPP theorem). Additionally, choreographies prevent any mismatch between `send` and `receive` operations by construction, since the language constructs for interaction describe the action of both parties. This in turn implies deadlock-freedom, which extends to any projected implementations thanks to the EPP theorem. This combination of features and guarantees is what makes choreographies a good candidate for describing communicating systems, and an interesting subject for our verification efforts.

Verified stacks aim to produce a set of tools and techniques—involving a combination of interactive theorem provers, SMT-solvers, and other tool-chains—that allow users to describe programs, and through a mechanized process, generate a representation in a target language (usually binary) while providing some formal guaranties about the behaviour of the resulting program w.r.t. the initial specification. Some examples of verified stacks are compilers like CakeML [4] and CompCert [5]. They provide a proof of correctness that guarantees that the observable behaviours of the generated executables are compatible with the behaviours of the source programs. Other approaches, like the seL4 microkernel [3], target specific programs and can include proofs of additional properties like security or deadlock-freedom.

By connecting a choreography language with a verified stack, one could provide end-to-end guarantees about communicating systems, using a simple and expressive interface with convenient properties. However, most EPP results in the literature target modelling languages like the π-calculus rather than concrete programming languages, and make no attempt to extend the results to executable representations of the protocol participants. Furthermore, there is limited support for representing (concurrent) interactions in most verified stacks, which restricts their usability to only sequential programs. We aim to address these issues by providing i) to the best of our knowledge, the first mechanized proof of the EPP theorem, ii) a proven correct choreographic compiler, and iii) a novel approach for dealing with open system specifications.

Our implementation uses the HOL4 theorem prover and is comprised of a choreographic language definition that gets projected using EPP to endpoints expressed in a process algebra which is similar to value-passing CCS, but also features explicit locations, internal and external choice, and a more concrete representation of data and local computations that is amenable to code generation. From there, a sequence of verified refinements of the endpoints gradually translates each process into concrete CakeML code. The first step eliminates the internal and external choice primitives by encoding them using send, receive and if-statements. The second step compiles from a process algebra where messages can have unlimited length to a process algebra where messages have a (configurable) maximum payload size by introducing a protocol which sends messages in smaller chunks. The (by now sequential) code at each location is translated into a functional program in the CakeML language. Finally, we can use the verified CakeML compiler [8] as a backend to compile the functional representation into concrete machine code for mainstream architectures (including x86-64 and ARMv8) such that the

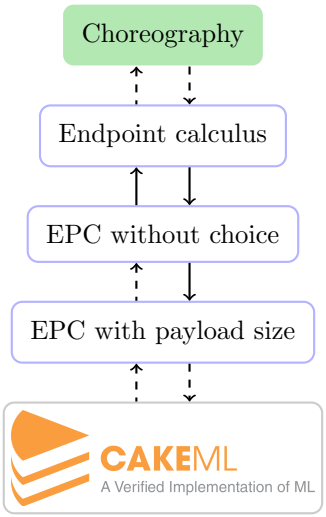


Figure 2: Roadmap

machine code preserves the observable behaviour of the CakeML program. By connecting the correctness proofs for each of these intermediate steps, we can obtain a top-level correctness statement that the communication behaviour and deadlock freedom properties of the choreography are preserved down to the machine code that runs it.

Local computations (e.g: `price(x)` in Figure 1) are shallowly embedded as functions in higher-order logic, hence can be directly translated by CakeML's proof-producing synthesis tool [7], or left underspecified to model external components. Finally, the underlying `send` and `receive` operations are implemented using CakeML's foreign function interface, which allows their use to be back-end agnostic (sockets, IPC, etc).

A high-level roadmap of our work is described in Figure 2, where each arrow signals a proof of semantic correspondence between the intermediate representations. Downwards arrows are semantic preservation proofs, and upward arrows are semantic reflection. Dashes account for sections pending verification. Additionally, proofs of confluence, deadlock-freedom, and structural congruence laws are in place for the semantics of our choreography language, and an extension for the CakeML FFI is being develop to allow for parameterized `send` and `receive` specifications.

In conclusion, by building on top of a flexible and robust verified stack like CakeML, and taking advantage of the strong guarantees and ease of use of choreographies, we believe this work will make the task of developing and maintaining verified systems a much more simple and cost-effective one.

References

[1] Laura Bocchi, Hernán Melgratti, and Emilio Tuosto. Resolving non-determinism in choreographies. In *European Symposium on Programming Languages and Systems*, pages 493–512. Springer, 2014.

[2] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014.

[3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[4] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ml. In *ACM SIGPLAN Notices*, volume 49, pages 179–191. ACM, 2014.

[5] Xavier Leroy et al. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[6] Fabrizio Montesi. Choreographic programming, 2014.

[7] Magnus O. Myreen and Scott Owens. Proof-producing synthesis of ML from higher-order logic. In *Proceedings of ICFP*, pages 115–126, 2012.

[8] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony C. J. Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *Proceedings of ICFP*, pages 60–73, 2016.

Towards Compositional User Interfaces: Semantics of Multi-Way Dataflow Constraints

Magne Haveraaen and Jaakko Järvi*

University of Bergen, Norway
{magne.haveraaen|jaakko.jarvi}@ii.uib.no

Introduction

In today's dominant user interface programming paradigm, programmers write handler functions to respond to user interaction events. They must coordinate responses so that UI behavior is consistent in all possible event orderings and timings. This is recognized as a complex, costly, and difficult programming task [3]. Further, since UI logic has no tangible representation as an algorithm or a component, code that manages user interactions is not reusable.

Our prior work on UI programming [1] shows that user interface behaviors can be expressed as reusable algorithms, parameterised over a *multi-way dataflow constraint system*. Our vision is a programming model, where GUI fragments are components, and GUIs are compositions of these fragments, each defined by a constraint system. The composition of UIs thus builds on the composition of constraint systems, which is the focus of this paper: an investigation of semantical underpinnings of multi-way constraint systems. The goal is a formal framework to serve as a basis for designing (GUI) algorithms over constraint systems and for modular reasoning about constraint systems, a step towards compositionality in GUI programming.

Background: dataflow constraint systems in GUIs

A *dataflow constraint* describes a relation amongst variables and means to satisfy that relation. The latter is a set of functions, *constraint satisfaction methods*, that compute values of some of the constraint's variables, using others as inputs. A collection of constraints to be satisfied simultaneously is a *constraint system*. There are many variations of dataflow constraint solvers and systems [5, 4, 6], and many uses, most prominently perhaps GUI widget layout. Our use of constraints in GUIs is described in [1].

Figure 1 shows an idealized GUI for resizing an image, which we use as an example. The image's initial width and height are determined at the launch of the dialog. The user can specify a new width and height relative to the initial values, or directly as the number of pixels. Further, the user can request that the GUI preserves the initial aspect ratio.

The variables are dependent on each other: changing the value of one triggers changes in others, so that the GUI returns to a consistent state. The dependencies and the consistent

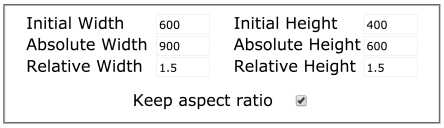


Figure 1: An example GUI implemented using a constraint system.

*This work was supported in part by NSF grant CCF-1320092.

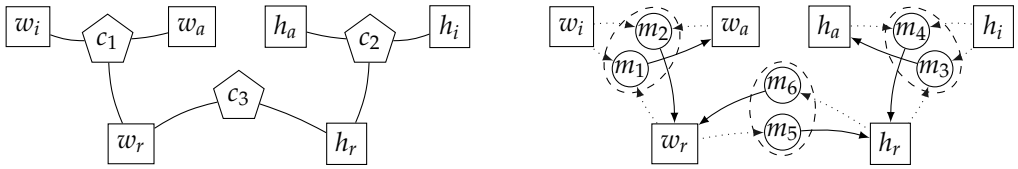


Figure 2: The constraint system arising from the GUI of Figure 1. Figure a depicts the relations (c_1, c_2, and c_3) amongst the variables in the system. Figure b shows the relations split into functional dependencies. A dashed ellipsis marks the set of methods that together implement a constraint's relation; for example, applying either of the methods m_3 or m_4 satisfies c_2.

states are expressed by the constraint system in Figure 2. The variables w_i, w_r, and w_a are bound, respectively, to the initial, relative, and absolute width fields, and h_i, h_r, and h_a to the corresponding height fields. Figure 2a shows the three constraints in the system: c_1 enforces the relation $w_a = w_i * w_r$, c_2 the relation $h_a = h_i * h_r$, and c_3 the relation $w_r = h_r$. The c_3 constraint is only active if the initial aspect ratio should be retained. These relations are decomposed into sets of functional dependencies (*methods*), shown in Figure 2b. Solving a *dataflow constraint system* boils down to selecting one method from each constraint such that they can be executed in an order that does not invalidate already enforced constraints, and then executing them.

In practice the constraints' relation is implicit, defined indirectly by the constraint satisfaction methods that the programmer writes. Questions about the composability of constraints reduce to questions about the graphs formed by connecting methods via their common variables—and they ignore the semantics of the constraints. Hence this paper.

Syntax and Semantics of Constraints and Constraint Systems

The programming language for constraint systems is defined with set theoretic semantics for an interface, a set of declarations of types and functions. The basic syntactic module is an interface. An *interface* I declares a set of *types* $\mathrm{Typ}(I)$ and a set of operations (or functions) $\mathrm{Fun}(I)$ in the usual way. The *expressions* of type $t \in \mathrm{Typ}(I)$ on an interface I with variables V is a set $E_{I,V,t}$, generated in the usual way. We denote the set of all type correct expressions as $E_{I,V} = \uplus_{t \in \mathrm{Typ}(I)} E_{I,V,t}$, and extend $\mathrm{typ} : \mathrm{Nam}(V) \to \mathrm{Typ}(I)$ to a function $\mathrm{typ} : E_{I,V} \to \mathrm{Typ}(I)$. A *substitution on variables* $s : V \to E_{I,X}$ is a function from $\mathrm{Nam}(V)$ to $E_{I,X}$ such that $\mathrm{typ}(v) = \mathrm{typ}(s(v))$ for all $v \in \mathrm{Nam}(V)$. A substitution on variables $s : V \to E_{I,X}$ extends to a substitution on expressions $s' : E_{I,V} \to E_{I,X}$ by recursing to subexpressions. The composition of a substitution $r : X \to E_{I,Y}$ and $s : Y \to E_{I,Z}$ is a substitution $r;s : X \to E_{I,Z}$ defined by $r;s(x) = s'(r(x))$, for all $x \in X$.

The *model* $\mu : I \to \mathbf{Set}$ of an interface I defines a set $\mu(t)$ for every $t \in \mathrm{Typ}(I)$ and a total function $\mu(f) : \mu(t_1) \times \cdots \times \mu(t_k) \to \mu(t)$ for every function $f : t_1, \ldots, t_k \to t \in \mathrm{Fun}(I)$. An *allocation* $a : V \to \mu$ of values to variables V for an interface I with model μ defines a value $a(v) \in \mu(\mathrm{typ}(v))$ for all $v \in \mathrm{Nam}(V)$. We define the semantics $[\![\_]\!]_{\mu,a,t} : E_{I,V,t} \to \mu(t)$ of expressions of type $t \in \mathrm{Typ}(I)$ and allocation a in the usual way.

Constraints formulate requirements on allocations of variables. An *equational constraint* on variables V over an interface I is a pair $e_1 = e_2$ where $e_1, e_2 \in E_{I,V}$ and $\mathrm{typ}(e_1) = \mathrm{typ}(e_2)$. Let μ be a model for I. The equational constraint $e_1 = e_2$ holds for an allocation $a : V \to \mu$ iff $[\![e_1]\!]_{\mu,a} = [\![e_2]\!]_{\mu,a}$.

Let I be an interface and V a collection of variables for I. The *constraints* on variables V over I is a set $C_{I,V}$ generated by an equational constraint $e_1 = e_2$ on variables V over an interface I,

where $e_1, e_2 \in E_{I,V}$ and typ(e_1) = typ(e_2); a conjunction $c_1 \& c_2$, disjunction $c_1 | c_2$, and negation $\neg c_1$ of a constraints $c_1, c_2 \in C_{I,V}$; and constant constraints FALSE and TRUE.

Let μ be a model for I. For a given constraint $c \in C$ and an allocation $a : V \to \mu$, the semantics of c, $[\![c]\!]_{\mu,a}$ defines when c holds or not. When c is an equational constraint $e_1 = e_2$ then $[\![c]\!]_{\mu,a} = ([\![e_1]\!]_{\mu,a} = [\![e_2]\!]_{\mu,a})$; when a conjunction $c_1 \& c_2$ then both c_1 and c_2 must hold for c to hold; when a disjunction $c_1 | c_2$ then at least one of c_1 and c_2 must hold for c to hold; etc.

A substitution $s : V \to E_{I,X}$ can be applied to a constraint $c \in C_{I,V}$. Then $s(c) \in C_{I,X}$ is the constraint obtained by replacing each expression e in c by $s(e)$.

In this abstract we state without elaborating that constraints, allocations, and models can be put together to form an *institution* [2] where the signatures are sets of typed variables and the signature morphisms are substitutions. Institution theory can be interpreted as a framework for modular reuse of specification and code. We use this interpretation to provide a module framework for constraint systems, and achieve flexible composition of constraints.

Example 1. The two constraint system primitives in the image scaling GUI, one between w_a, w_i, and w_r, and the other between h_a, h_i, and h_r, can be constructed from the same constraint system primitive (we assume $v_i \neq 0$) $G = \langle \{v_i : \mathbf{int}, v_a : \mathbf{int}, v_r : \mathbf{float}\}, \lfloor v_i v_r \rfloor = v_a, \{(v_a := \lfloor v_i v_r \rfloor), (v_r := v_a / v_i)\} \rangle$, using the substitutions $s_w : \{v_i, v_a, v_r\} \to \{w_i, w_a, w_r\}$ and $s_h : \{v_i, v_a, v_r\} \to \{h_i, h_a, h_r\}$. Their composition is a constraint system primitive whose constraint defines the GUI's consistent states and whose methods provide the means to bring the GUI into such a state:

$$\begin{aligned} s_w(G) \,\&\, s_h(G) = \langle \{&w_i : \mathbf{int}, w_a : \mathbf{int}, w_r : \mathbf{float}, h_i : \mathbf{int}, h_a : \mathbf{int}, h_r : \mathbf{float}\}, \\ &\lfloor w_i w_r \rfloor = w_a \;\&\; \lfloor h_i h_r \rfloor = h_a, \\ &\{(w_a := \lfloor w_i w_r \rfloor), (w_r := w_a / w_i), (h_a := \lfloor h_i h_r \rfloor), (h_r := h_a / h_i)\} \rangle. \end{aligned}$$

To obtain a constraint system primitive for the same GUI but requiring equal scaling for width and height, we use substitutions $r_w : \{v_i, v_a, v_r\} \to \{w_i, w_a, a\}$ and $r_h : \{v_i, v_a, v_r\} \to \{h_i, h_a, a\}$:

$$\begin{aligned} r_w(G) \,\&\, r_h(G) = \langle \{&w_i : \mathbf{int}, w_a : \mathbf{int}, h_i : \mathbf{int}, h_a : \mathbf{int}, a : \mathbf{float}\}, \\ &\lfloor w_i a \rfloor = w_a \;\&\; \lfloor h_i a \rfloor = h_a, \\ &\{(w_a, h_a := \lfloor w_i a \rfloor, \lfloor h_i a \rfloor), (w_a, a := \lfloor w_i (h_a / h_i) \rfloor, h_a / h_i), \\ &\;(h_a, a := \lfloor h_i (w_a / w_i) \rfloor, w_a / w_i)\} \rangle. \end{aligned}$$

References

[1] Gabriel Foust, Jaakko Järvi, and Sean Parent. Generating reactive programs for graphical user interfaces from multi-way dataflow constraint systems. In *Proc. of the 2015 Int. Conf. on Generative Programming: Concepts and Experiences*, GPCE 2015, pages 121–130, New York, NY, USA, 2015. ACM.

[2] Joseph A. Goguen and Rod M. Burstall. Institutions: abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.

[3] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, Carnegie Mellon University Computer Science Department, July 1993.

[4] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad V. Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, November 1990.

[5] Ivan E. Sutherland. Sketchpad: a man-machine graphical communication system. In *Proc. of the May 21-23, 1963, Spring Joint Computer Conf.*, AFIPS '63, pages 329–346, New York, NY, USA, 1963. ACM.

[6] Brad Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, January 1996.

Specifying with Syntactic Theory Functors

Magne Haveraaen[1] and Markus Roggenbach[2],

[1] Universitetet i Bergen, Norway
[2] Swansea University, Wales, UK

Goguen & Burstall institution theory [1] captures the structure of many approaches to software and information systems, e.g., ontologies, modeling and formal specifications. Institution theory also provides powerful institution independent structuring mechanisms, allowing extensive reuse within and between institutions. Such structuring is considered 'good practice' for many reasons, including separation of concerns, ease of reuse of specification-text, and improved theorem proving support.

However, solving practical problems still hits issues of structuring and reuse beyond the classical mechanisms. Though immaterial to foundational theory, lack of support causes lengthy writing of boilerplate code or repeated adaptation from one context to another. Take for example the 'challenge' to specify in first order logic that a sort s has at least five elements, presented here in CASL [4] syntax:

spec SORTWITHMINIMUM5ELEMENTS =
  sort s
  ops $c1, c2, c3, c4, c5 : s$
  • $\neg\ c1 = c2 \wedge \neg\ c1 = c3 \wedge \neg\ c1 = c4 \wedge \neg\ c1 = c5 \wedge$
    $\neg\ c2 = c3 \wedge \neg\ c2 = c4 \wedge \neg\ c2 = c5 \wedge$
    $\neg\ c3 = c4 \wedge \neg\ c3 = c5 \wedge$
    $\neg\ c4 = c5$
end

We observe that writing out this standard pattern of axioms becomes awkward for larger numbers, and may easily introduce errors in a conceptually simple specification. Thus, we propose that specifications involving such patterns should automatically be generated. More concisely, we claim that, like in programming, also in specification there are recurring problems for which there exist 'standard solutions.' These can and ought to be addressed by 'patterns.'

In order to be useful, one would require such solutions to be 'flattenable,' i.e., to be syntactic sugar. This would allow the specifier to expand a 'macro,' possibly with tool support, in order to check if the generated axioms actually look as expected. Furthermore, it would appear natural for such a pattern to take selected signature elements (e.g., the sort and a list of constants for defining any number of distinct constants) as their arguments. Such arguments would 'steer' the patterns' use in the concrete application context. For example, we can lift a set of functions on elements to functions on arrays. An n-ary function would lift to a function on n arrays. To specify the effect of lifting, the lifting pattern needs to identify the element type and set of functions to be lifted, as well as the array type, the index type and function.

In recent work [2], we proposed a framework for creating institution specific structuring mechanisms, *syntactic theory functors* (STFs). STFs subsume the standard institution independent structuring mechanisms, and open up new ways of reusing existing and structuring new specifications. This allows building a rich institution incrementally from a simple institution and appropriate STFs, very much needed when developing ontologies or specification languages for a new domain. Richer institutions yields more versatility to the Distributed Ontology, modeling and specification Language (DOL), which provides structuring and interoperability between any institution based formalism.

Definition 1 (Syntactic Theory Functor (STF))**.** *We call a functor F on a theory (Σ, Φ) consisting of a signature Σ and a set of logical formulae Φ to be* syntactic *if the application of F can be decomposed into component expressions, namely*

- $F_{\mathtt{sig}} : \mathbf{Sig} \to \mathbf{Sig}$ *is the effect of F on the signature, i.e., $F_{\mathtt{sig}}(\Sigma) = F(\Sigma, \Phi)_1$,*
- $F_{\mathtt{base}} : \Sigma \to \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$ *is the base axiom giving the properties of the constructed specification for a signature Σ, and*
- $F_{\mathtt{for}} : \mathrm{for}(\Sigma) \to \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$ *is the effect of F on each formulae on the signature Σ.*

STFs exhibit a number of useful properties, including

Theorem 1 (Elementary properties of STFs)**.**

- *Syntactic theory functors are additive, i.e., $F(\Sigma, \Phi_1 \cup \Phi_2) = F(\Sigma, \Phi_1) \cup F(\Sigma, \Phi_2)$.*
- *The identity functor on theories is a syntactic theory functor.*
- *Syntactic theory functors compose.*

Concerning our above example, the distinctness of a given set of constants can be defined by an STF called $\mathrm{Free}_{s,C}$, where s is a sort and C is the desired set of distinct constants, all of type s:

$$\begin{aligned}
\mathrm{Free}_{s,C,\mathtt{sig}}(\Sigma, \Phi) &= \Sigma \cup \langle \{s\}, C, \emptyset, \emptyset \rangle, \\
\mathrm{Free}_{s,C,\mathtt{base}}(\Sigma, \Phi) &= \bigcup_{c,c' \in C, c \neq c'} \{c \neq c'\}, \\
\mathrm{Free}_{s,C,\mathtt{for}}(\Sigma, \Phi) &= \Phi.
\end{aligned}$$

Using this functor, we can now alternatively specify sort s to have at least five elements, presented here in CASL syntax augmented by STFs:

$$\mathrm{Free}_{s,\{c1,c2,c3,c4,c5\}}(\{\})$$

where $\{\}$ denotes the empty CASL specification. This specification using the Free STF is semantically equivalent to SORTWITHMINIMUM5ELEMENTS. Note that there are further STFs available in order to solve this problem, e.g., $\mathrm{Free}'_{s,n}$, which takes the number n of elements required in sort s as a parameter and does not require the specifier to provide names of constants that shall denote the elements present in the carrier set. The natural advantage of Free′ over Free is that it requires less to write by the specifier, however, often one will want to know the concrete names of the constants involved.

In our NWPT contribution we will demonstrate that systematic use of STFs makes writing specifications easier, as there is (1) less text to write and (2) the specifier can concentrate on the essential, new new things to be considered. Furthermore, using STFs makes reading and comprehending of specifications easier, as (1) specifications become shorter and (2) standard constructions are named as such, i.e., the reader can concentrate on the fundamentally new, non-standard elements.

To this end, we will further develop the concepts presented in [2] and, in particular, we will experiment with and analyse STF specification practice. Here, we will study various approaches concerning partiality (error algebras, ok-predicates, ...) and order sorting as surveyed by [5], the technique of "plainification" which uses preconditions for functions in algebraic specifications [3], and also present concrete examples such as specifying money conversion and analyse how STFs can help writing 'better' specifications.

References

[1] J.A. Goguen, R.M. Burstall, *Institutions: Abstract Model Theory for Specification and Programming.* Journal of the ACM, 1992.

[2] M. Haveraaen, M. Roggenbach, *Syntactic Theory Functors.* Abstract, presented at WADT 2018.

[3] M. Haveraaen, E.G. Wagner, *Guarded Algebras: Disguising Partiality so You Won't Know Whether It's There.* WADT 1999: 182-200.

[4] P. Mosses, *CASL Reference Manual: The Complete Documentation Of The Common Algebraic Specification Language.* Springer, 2004.

[5] P. Mosses, *The Use of Sorts in Algebraic Specifications.* COMPASS/ADT 1991: 66-92.

Static analysis for dynamic data race detection with TeSSLa *

Svetlana Jakšić[1], Malte Schmitz[2], Volker Stolz[1], and Daniel Thoma[2]

[1] Western Norway University of Applied Sciences, Bergen, Norway
{svetlana.jaksic, volker.stolz}@hvl.no
[2] Institute for Software Engineering and Programming Languages, Universität zu Lübeck, Germany
{malte.schmitz, thoma}@isp.uni-luebeck.de

Abstract

We investigate the use of hardware-based tracing facilities for the purpose of detecting potential data races. Modern processors offer advanced facilities for introspection of *control flow*, such as the Intel Processor Trace (IntelPT), or the CoreSight TraceMacrocell on ARM processors. Information about *data* such as memory addresses has to either be logged explicitly by the running code through additional trace packets in the hardware trace (only available on ARM), or on a different software-based channel.

Due to the high trace data volume and the need for instrumentation, we present here our approach for minimising the amount of necessary instrumentation that our stream-based implementation of the lockset algorithm requires, through the help of static analysis.

Multicore processors have become commonplace. They offer speed boosts in two ways in modern IT: on the one hand, different tasks can run concurrently on multiple cores, sharing operating system resources. On the other hand, developers have the option of tailoring their applications to make use of multiple cores by explicitly introducing concurrency, usually in the form of threading. These threads by their very definition have the possibility to affect each other: memory is shared between threads, and synchronization APIs allow for higher levels of abstraction. For programs, for example, written in the C programming language, a careful combination of modification of shared data and locking/synchronization is necessary.

Using those concurrency constructs and their interactions incorrectly leads to widely discussed problems, such as deadlocks [1] and data races [4]. While the first type of problem is easily detectable at runtime, the latter is more insidious. Traditionally, these types of software defects have been approached from two sides: *static analysis* checks the source code and reports potential errors. To that end, over-approximations of the behaviour of the program are used (e.g. in terms of variable accesses and lock operations), which may lead to uncertainties on whether a particular behaviour will actually occur during runtime due to general issues on decidability. In contrast to analyzing the code before it actually runs, *dynamic analysis* looks at the actual behaviour of a program. Although this only gives a limited view on the behaviour of the actually executed code, it allows for the precise reporting of actual occurrences, which can be used to predict potential erroneous behaviour across different runs.

Here, we present an approach to dynamic data race monitoring that reduces the amount of instrumentations required by using hardware-based tracing facilities. As these facilities only provide control-flow information, but no information about the manipulated data, they cannot replace instrumentation in general. At runtime, we then combine both sources of information to run the *lockset algorithm* [5] and detect potential for data races. The algorithm itself is expressed and executed in the stream-based specification language TeSSLa [3] which provides a suitable match for the events generated by the instrumentation and the hardware.

*This work was partially supported by the European Horizon 2020 project COEMS under grant agreement no. 732016 (https://www.coems.eu/).

Dynamic data race detection with TeSSLa. The standard definition of a data race is as follows [5]: a data race in the multi-threaded program exists when two or more threads access the same memory location concurrently, at least one of the accesses is a write and the threads are not using any synchronization mechanism to control their accesses to that memory. One of the most frequently used algorithms for finding possible data races is the lockset algorithm, introduced and implemented in Eraser [5]. We have chosen this algorithm because it is commonly used by the state-of-the-art static [6] and dynamic tools [7] and it is suitable for implementation in TeSSLa. The lockset algorithm finds data races by checking if a program follows a certain locking discipline. In particular, it enforces that every shared variable is always protected by some set of locks and this set of locks must be held by any thread accessing the variable. The lockset algorithm needs relevant events with respect to *locking* (which thread is taking which lock), and *memory accesses* (which thread is reading from or writing to a memory location).

In [2], we have shown how to specify the lockset algorithm in the stream-based specification language TeSSLa. We use a software-based instrumentation, which works very much along the lines of other runtime checkers, such as ThreadSanitizer [7], using the LLVM Compiler Infrastructure. Every event that the software instrumentation generates, contains at least the information about source code location, type of access (variable read/write, function call, lock or thread operation), and the identity of the executing thread. Depending on the type, the event is augmented with information such as the memory location for a read or write access, any offset and size of the access, and lock operations.

This poses the challenge that memory accesses are almost indiscriminately instrumented, leading to a high volume of events that the dynamic analysis has to process. As instrumentation has overhead, minimizing the number of instrumentations without compromising race detection would thus improve performance of the application being monitored.

Reducing dynamic overhead through static analysis A feature of modern mainstream CPUs such as Intel and ARM is now a built-in tracing facility that goes beyond performance counters that have traditionally been found on CPUs. High-volume trace data about the executed instructions are made available to monitoring tools. To achieve a high data rate, this instruction trace is compressed in the following way (simplifying a bit, eliding synchronization frames etc.): only on encountering a conditional branch an event is generated in the form of a single bit, indicating whether the branch has been taken or not. Additional information may be included e.g. for function calls and returns. This trace is then stored e.g. in a ring buffer and exported, and has for example been used in a sampling-based approach to race detection [8].

The trace in the compressed format is only intelligible with the corresponding binary (object code), as events have to be mapped to the corresponding assembler statements, and possibly instructions in between two events have to be synthesized. Most notably, any data (values) that is not hard-coded into the binary (such as constants being loaded into registers), is absent.

For our data race detection, that means we need to either reconstruct vital information from the trace, or use instrumentation in addition when this is not possible. An example target for reconstruction are accesses to global variables: accesses to them (and e.g. arrays) can be generalized to first loading a base address, and then having the actual memory access with a constant offset to this base, and are hence prime candidates for reconstruction from the trace. Indirect accesses to dynamically allocated memory (through pointers) however pose a challenge, as the base is not a constant. A further complication are obtaining arguments to function calls such as the locking/unlocking operations.

Contribution. Here, we optimise the full instrumentation of memory accesses and lock-operations presented in [2] by eliding event-generation, when those events can be reconstructed from the control flow trace. During the preprocessing phase, we derive static information about the data for each of the relevant operations (Load/Store/Lock/Unlock). In those cases where our conservative approximation yields a unique result, i.e. where the data belonging to an operation is statically decidable, we do not emit an instrumentation, but rather store which generated instruction-fragment has which effect in terms of the lockset algorithm for subsequent use during the dynamic phase.

Then, during execution, we receive an interleaved stream of events from the instrumentation, and instruction-trace events from which we reconstruct the corresponding lockset-event. A technical challenge in the current Intel Processor Trace (Intel PT)-based setup is that we receive instrumentation events and control-flow events on separate, unsychronized streams.

This combination should allow us to avoid some of the performance penalties of purely software-based data race checkers such as Thread Sanitizer [7], at the cost of some static analysis up front, as well as moving the data race checking out of the context of the running program.

References

[1] E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.

[2] S. Jakšić, D. Li, K. I. Pun, and V. Stolz. Stream-based dynamic data race detection. In *NIK 2018*. NIK OJS, 2018. http://ojs.bibsys.no/index.php/NIK/article/view/511.

[3] M. Leucker, C. Sánchez, T. Scheffel, M. Schmitz, and A. Schramm. TeSSLa: Runtime verification of non-synchronized real-time streams. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018*, pages 1925–1933. ACM, 2018.

[4] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Letters on Programming Languages and Systems*, 1(1), Mar. 1992.

[5] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[6] H. Seidl and V. Vojdani. Region analysis for race detection. In *Proceedings of 16th International Symposium on Static Analysis (SAS)*, volume 5673 of *Lecture Notes in Computer Science*, pages 171–187. Springer, 2009.

[7] K. Serebryany, A. Potapenko, T. Iskhodzhanov, and D. Vyukov. Dynamic race detection with LLVM compiler - compile-time instrumentation for threadsanitizer. In *Proceedings of the Second International Conference on Runtime Verification, RV 2011*, volume 7186 of *Lecture Notes in Computer Science*, pages 110–114. Springer, 2011.

[8] T. Zhang, C. Jung, and D. Lee. Prorace: Practical data race detection for production use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, pages 149–162. ACM, 2017.

Contract-based Verification of Asynchronous Calls with Cooperative Scheduling — Extended Abstract

Eduard Kamburjan[1], Crystal Chang Din[2],
Reiner Hähnle[1], and Einar Broch Johnsen[2]

[1] Department of Computer Science, Technische Universität Darmstadt, Germany
{kamburjan,haehnle}@cs.tu-darmstadt.de
[2] Department of Informatics, University of Oslo, Norway
{crystald,einarj}@ifi.uio.no

Introduction. Method contracts [6] enable compositional reasoning about method calls in a synchronous setting. However, in an asynchronous setting the called object may be executing other processes between the call time and the execution time of a method. The generalization of method contracts to asynchronous communication in a concurrent setting is not well understood. In this paper we generalize method contracts for synchronous, sequential execution to asynchronous method calls in concurrent *Active Object* languages [4].

Synchronous Method Contracts. Method contracts are an established formalism for specification and deductive verification of object-oriented languages (e.g., [3, 6]). These contracts are abstractions of method executions and describe (1) the context a method relies on as a precondition and (2) the guarantees the method provides as a postcondition. In a synchronous setting, a precondition pre describes the heap and the method parameters at the moment of the call. A postcondition post describes the heap (including any side effects of method execution on the heap) and the return value at termination. To show that a method `m` satisfies a method contract $(\text{pre}, \text{post})$, we prove that if pre holds at the point when `m` is called, then post holds immediately after `m` terminates.

Active Objects. Active objects with asynchronous method calls use a more complex call mechanism than context switches, which gives rise to four challenges:

1. **Call Time Gap.** There is a delay between the execution of a call statement and the start of the execution of its associated process. During this delay, the called object ("callee") may execute[1] other processes. Even if a precondition holds at method invocation time, it does not necessarily hold at activation time when the method actually begins to execute.
2. **Strong Encapsulation.** Each object has exclusive access to its fields. The caller object cannot access fields of the callee; and is thus not able to evaluate the precondition of an invoked method if the precondition depends on the fields of the callee.
3. **Interleaving.** In concurrent active object languages, processes may interleave at suspension points. At these points it is challenging to know which properties about the heap can be relied on and which must be guaranteed.
4. **Return Time Gap.** Active objects use futures to decouple method calls from reading the return value. A future is a "mailbox" generated and associated with the asynchronous call. To read the return value, an object synchronizes with the future, which can also be shared with other objects. Another object which reads the future, need not know which method was used to resolve the future. It is generally not possible to know which postcondition held at the time when a future was resolved.

[1] For Active Objects, only one process is active in an object at any given time.

Asynchronous Method Contracts. To address challenges (1) and (2), we split the precondition into a *parameter* precondition and a *heap* precondition. Method parameters cannot be changed by other objects and their values are by definition accessible to the caller. Therefore, the part of the precondition that talks about method parameters can and must be established by the caller. On the other hand, the caller cannot know which other processes might change the heap before the method starts to execute and even if it would know, it could not access the fields of the callee. The heap precondition must be established by the *last active* process executed on the callee prior to the actual method execution and can only be guaranteed by the callee. The callee needs to describe the potentially active processes between the method call and the method execution. We extend heap preconditions with a *concurrency context* consisting of two *context sets* with methods: methods that may run in parallel with the considered method and those that must have finished before the considered method can be executed.

To address the remaining challenges, we extend the possible scope of contracts from start and end points of method execution to any synchronization point. This is natural in an active object setting, because the effect of a method execution needs to be known and analyzed when interleaving occurs and when the return value is resolved, not when it is read.

For (3), we introduce *suspension* contracts that specify the behavior of the contract during suspension and contain the methods possibly ran last during suspension and a condition for their last state: During suspension other tasks may be executed on the object and, since they share the same memory, data races are possible. However, these data races are *localized* and can *only* occur at method start and suspension points. This is due to that all ABS methods run uninterruptedly either to completion or suspension. Only the *final state* at these points is relevant when analyzing local data races. This justifies to specify active objects with generalized synchronous contracts, rather than using a general purpose mechanism, e.g., separation logic.

For (4) we specify a *resolve contract* at each synchronization point, which contain the set of methods that are permitted to resolve a future. This abstract only presents our approach to solve (1), (2). For full details and how to deal with (3), (4) we refer to our report [5].

Specifying State in an Asynchronous Setting As discussed above, we split the precondition of asynchronous method contracts into a parameter precondition and a heap precondition. The *parameter precondition* is guaranteed by the caller, who knows the appropriate synchronization pattern. It is part of the interface declaration of the callee and exposed to the world. The *heap precondition* is guaranteed by the callee and is declared in a class and not exposed.

The indirect control of the caller over the callee's state via method calls is part of the control flow of the class: A method is expecting a certain *concurrency context*, in addition to its memory context. A concurrency context is part of the contract as two *context sets* containing methods:

- Each of the methods in *succeeds* must guarantee the heap precondition and at least one of them must have run before the specified method starts execution.
- Each of the methods in *overlaps* must preserve the heap precondition of the specified method. Between the termination of the last method from *succeeds* and the start of the execution of the specified method, only methods from *overlaps* are allowed to run.

As a default, these contexts contain all methods. In this case the heap precondition degenerates into a class invariant and has to be guaranteed by every process at each suspension point.

Example 1. *Consider the following interface and class declaration. Class* `C` *has an internal counter* `r` *and declares a method* `m` *that divides its parameter by the counter. The counter must*

be strictly positive and must have been increased at least once before calling `m`*, while* `getVal` *can be read arbitrarily often before doing so. We use a JML-style [3] specification language.*

```
1 interface I {
2   /*@ requires True; @*/
3   Unit n();
4   /*@ requires True; @*/
5   Int getVal();
6   /*@ requires i > 0; @*/
7   Rat m(Int i);
8 }
```

```
9 class C(Int r) implements I {
10  /*@ requires r >= 0; @*/
11  Int n() { r = r + 1; }
12  Int getVal() { return r; }
13  /*@ requires r > 0;
14      succeeds {n}; overlaps {getVal}; @*/
15  Rat m(Int i) { return i/r; }
16 }
```

The code fragments below represent different main blocks using a `C` instance `c`. The left fragment fails to verify the context sets specified above: Due to reordering, method `m` can be executed first, so condition `succeeds{n}` fails. The middle fragment fails as synchronizing only on `getVal` does not guarantee that `n` has terminated when `getVal` returns. The right fragment verifies.

```
1 c!n();
2 c!getVal();
3 c!m();
```

```
1 c!n();
2 await c!getVal();
3 c!m();
```

```
1 await c!n();
2 c!getVal();
3 c!m();
```

Postconditions and Propagation. As in JML, the postcondition is specified with a `ensures` clause in the contract of the methods, which is a formula over the fields and parameters of the class and method, as well as a special keyword for the return value. Contrary to JML, however, an additional step is used between specification and proof obligation generation: The context sets are used to propagate preconditions. Propagation adds the precondition of a method to the postconditions of all the methods in its `succeeds` sets and as invariants of all the methods in its `overlaps` sets. E.g., in the above example the contract of method `C.n` gets the additional clause `ensures r > 0;` from the context set `succeeds{n}` of `C.m`. This strengthening ensures that `C.m`'s precondition is established by all processes that can possibly run directly before it.

Verification. Pre- and postconditions are verified as a Dynamic Logic proof obligation, similar to the verification of JML contracts [1], but the context sets are verified by external static analyses: Must-Have-Happened and May-Happen-In-Parallel analyses [2] are used that the order of method calls specified in the context sets is ensured by the given program.

References

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification - The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.

[2] E. Albert, A. Flores-Montoya, S. Genaim, and E. Martin-Martin. May-Happen-in-Parallel Analysis for Actor-based Concurrency. *ACM Trans. Comput. Log.*, 17(2):11:1–11:39, 2016.

[3] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Proc. FMICS'03*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.

[4] F. S. de Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, Oct. 2017.

[5] E. Kamburjan, C. C. Din, R. Hähnle, and E. B. Johnsen. Contract-based verification of asynchronous calls with cooperative scheduling. Technical report, TU Darmstadt, 2018. http://formbar.raillab.de/en/techreportcontract/.

[6] B. Meyer. Design by contract: The Eiffel method. In *TOOLS (26)*, page 446, 1998.

The Future Mechanism and Information Flow Security

Farzane Karami[a], Christian Johansen[a], Olaf Owe[a], Gerardo Schneider[b]

[a] *Department of Informatics, University of Oslo*
[b] *Department of Computer Science and Eng., Chalmers University of Technology*

Introduction

Security for distributed systems is a critical issue since a large number of users and systems are affected by such systems. Challenges of information flow analysis of distributed systems depend on the communication paradigms used and their semantics. The *"actor paradigm"* [1] is a model advocated for designing distributed systems, in particular, it offers modular and compositional design and analysis. In addition to the actor model, the object-oriented paradigm has become popular because of its facilities for program structuring and reuse of code. These two paradigms are combined in the so-called *"active object"* model, where the objects are concurrent and autonomous, communicating with other objects by "asynchronous methods" [3].

We briefly discuss communication paradigms in active object languages using the syntax of the ABS (abstract behavioral specification) language [5]. A synchronous and blocking call of method m on a remote object o has the form $x := o.m(\overline{e})$ where $\overline{e}$ is the list of actual parameters. The caller is blocked until the callee returns the value, leading to unnecessary waiting. One way of avoiding blocking is achieved by using futures proposed in [2] and exploited in MultiLisp [4], ABCL [8], and several other languages [3]. A future is a read-only placeholder which eventually will contain the return value from an asynchronous method call [4, 7]. When a remote method call is made, a future object with a unique identity is created. The caller may continue with other computations while the callee is computing the return value. When the return value is computed, it is stored in the future object. The future is then said to be *resolved*. In a call statement $f := o!m(\overline{e})$, f is a *future variable* used to hold the future identity of the call, and the symbol "!" indicates an asynchronous method call. In the case of first-class futures, a future identity can be passed to objects desiring the return value of the method call, even before the value is computed. Thus, a future can be distributed to many active objects in a system.

The prevalence of futures in active object languages [3] highlights the significance of investigating inherent security and privacy issues related to futures. Futures might contain highly sensitive data, while in the case of first-class futures, any object that has a reference to it can access the content. Inside an object, high-sensitive data might be leaked from futures to low-secure variables,

Email addresses: farzanka@ifi.uio.no (Farzane Karami), cristi@ifi.uio.no (Christian Johansen), olaf@ifi.uio.no (Olaf Owe), gerardo@cse.gu.se (Gerardo Schneider)

and be transmitted to low-security actors, observable by an attacker. Consequently, it is critical to track futures and analyze their information flow security.

Information flow security challenges regarding futures

Future variables give a level of indirectness in that the retrieval of the result of a call is no longer syntactically connected to the call, compared to future-free languages. For instance when the future is received as a parameter, it may not statically correspond to a unique call statement. And different call statements may have different secrecy levels. One may overestimate the set of call statements that correspond to this given future parameter, but it requires access to the whole program. Therefore, when allowing futures as parameters, static information flow analysis would be imprecise, because the set of external calls that may result in an actual parameter is not statically given, and these calls are not uniform with respect to secrecy levels. In this case, one must consider the worst case possibility (i.e., the highest secrecy level) for the set of possible corresponding call statements, which is too conservative and severely limits statically acceptable information passing and call-based interaction, or requires dynamic checking.

In addition, the future concept comes with a notion of future identity, but not a notion of associated caller, callee. Identities of the caller and callee could in principle be incorporated in the future identity, but only at run-time. At static time there is no information about the caller and the creator of a future. This opens up for third party information with indirect/implicit handling of sensitive information. Static information flow regarding futures is too conservative. It causes unnecessary rejection of programs, especially when the complete program is not statically known as is usually the case in distributed systems. To address this problem, we propose an approach based on the notion of "*wrappers*" [6]. By wrapping futures and all objects receiving futures (recipient objects) as parameters or return values, we can prevent leakage of information from futures with high-sensitive data at the cost of dynamic checking.

We statically identify futures and recipient objects in the program. Then at dynamic time wrappers around them monitor and control communicated messages to or from these objects. When a low security object attempts to access a high secured future, the access should be rejected because of incompatible secrecy levels. The idea of wrappers is a permissive and precise dynamic approach, using the run-time environment to track information flow and monitoring the execution inside an object to prevent security violations. We modify the run-time environment, i.e., add another component to the environment to keep the secrecy levels of variables and change the operational semantic rules in a way to track flow-sensitive information flow dynamically. Fig. 1 exemplifies information flow analysis regarding futures and wrappers. An asynchronous method call toward object O creates future f, arrows 1, 2. Then it is passed to object P as a parameter of a method call. The future f, object O', and the recipient object P are wrapped by wrappers, marked in red, which checks whether it is safe to let them get the future value.

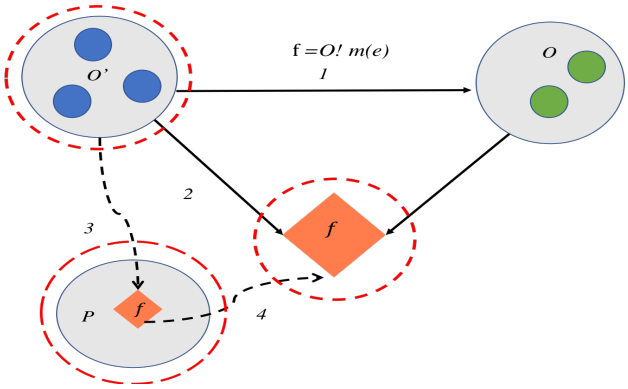


Figure 1: Information flow security regarding wrappers.

Conclusion

Futures are invented as a flexible way for sharing results and communication; however, their security and privacy are problematic. The notion of wrappers has been developed for safety of objects [6]. We here exploit wrappers for dealing with information security, by extending the runtime system with secrecy levels and apply dynamic checking for securing the use of futures.

References

[1] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, MIT Press Cambridge, MA, USA, 1986.

[2] H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, 1977.

[3] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, et al. A survey of active object languages. *ACM Computing Surveys (CSUR)*, 50(5):76, 2017.

[4] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[5] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Formal Methods for Components and Objects*, pages 142–164. Springer, 2011.

[6] O. Owe and G. Schneider. Wrap your objects safely. *Electronic Notes in Theoretical Computer Science*, 253(1):127–143, 2009.

[7] Y. Yokote and M. Tokoro. Concurrent programming in concurrent SmallTalk. In *Object-oriented concurrent programming*, pages 129–158. MIT Press, 1987.

[8] A. Yonezawa, editor. *ABCL: An Object-oriented Concurrent System*. MIT Press, Cambridge, MA, USA, 1990.

An Executable Modeling Language for Context-Dependent Self-Adaptive Systems

Sigurd Kittilsen[1], Jacopo Mauro[2], and Ingrid Chieh Yu[1]

[1] Department of Informatics, University of Oslo, Oslo, Norway
{sigurki, ingridcy}@ifi.uio.no
[2] University of Southern Denmark, Odense, Denmark
mauro@imada.sdu.dk

Extended Abstract

Self-adaptive systems are designed to deal with a continuously changing environment at runtime. They self-evaluate and adapt to meet new emerging situations that are not necessarily considered or planned before deployment, and continue their operations without the need for human supervision. It is expected that a self-adaptive system shall change its behavior in response to its perception of the environment and the system itself. The cause of the change may be internal causes (e.g., reached some failure state) or from the system's external context (e.g., changing weather condition or increasing service requests). The process of deciding how to react is often relevant to achieve a goal or objective that constitute the reason for building such a system. The demand for applications exhibiting self-adaptive properties is increasing in multiple domains, for example in Autonomous systems and IoT, Service-oriented computing, and agent systems. Software has been identified as the main enabling technology for achieving self-adaptivity and several challenges related to the designing and development of adaptive systems remain. Such challenges include finding a suitable level of abstraction to model and represent adaptive systems with corresponding environments [2, 3]. In particular, flexibility is one of the main concerns to achieve adaptation since hard-coded mechanisms make adapting long-running systems complicated. For this reason, there is a need for mechanisms that can dynamically specify the behavior in terms of high-level goals, following a more declarative approach rather than an imperative one. In addition to the modeling aspect, there is a need for analysis techniques to be able to give evidence that the system operates correctly and meets the desired expectation.

In our work, we try to address these challenges by looking into how engineering self-adaptive systems can benefit from formal methods and model-based engineering. Early in the development process, our approach aims at settling questions such as: *Is my goal specification precise enough to drive an effective adaptation? Is the chosen adaptation strategy for the application adequate? In what way does the non-deterministic and uncertain environment affect the adaptation of the system? In what range of environments is my goal reachable?*

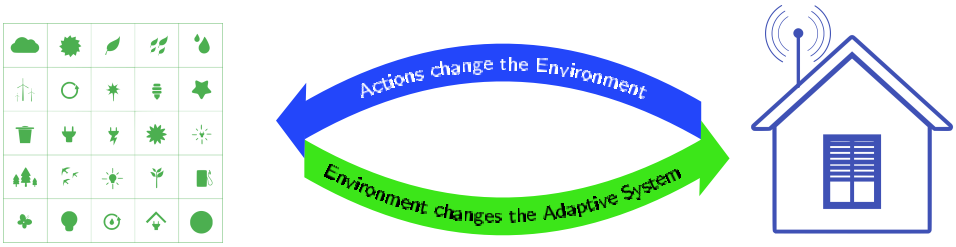


Figure 1: Self-adaptive system interacting with the environment.

As depicted in Figure 1, there is typically a two-way communication going on in self-adaptive systems. They behave differently depending on the state of the environment, and the environment is affected by the actions performed by the adaptive systems.

A well-known technique for enabling self-adaption is through the MAPE-k control loop [1], which will be essential to our work. MAPE-k is an acronym expressing the different stages in the loop. First, the adaptive system monitors (M) its environment through for example sensors. Then, the sensored data is analyzed (A), followed by a planning (P) phase that decides which actions to execute (E) depending on the gathered knowledge (k) and the system goals.

In this work, we propose a formal executable modeling language for self-adaptive systems. To achieve this, we extend Real-Time ABS [4], a modeling language that already combines actors with object-oriented structuring concepts and with cooperative concurrency, which allow complex synchronization between concurrent activities to be expressed. We also benefit from Real-Time ABS's support of the explicit modeling of deployment decisions for timed, resource-restricted models. However, ABS lacks language constructs to talk about important concepts in this area, like environments, system actions, and system-goals and the operational semantic does not consider the feedback processes typical for adaptive systems.

We will present the syntax and semantics of the extension, and show how the language supports simulation of self-adaptation to give predictions of applications' runtime adaptability. We will explain how we capture the communication between the adaptive system and the environment, and how the internal adaption of the system is modeled. The main idea is to integrate the MAPE-k loop into the operational semantics of the language. In this way, the execution flow will follow the different steps of the adaption loop while communication between the systems(s) and the environment keeps the monitored values up to date. We believe our design provides the developer with a high degree of modularity. The adaption-strategy as well as the different actions, and the environment itself can be replaced independently to simulate the system in different scenarios.

This is only the first step of our ongoing work. We plan on implementing our formal semantics into the back-ends of Real-Time ABS, which will enable the simulation of real use cases from our industrial partners in the Sirius-Center [5]. In the future, we will facilitate other types of analysis to complement the simulation. Examples of this could be static analysis to check for instance the reachability of goals. We also want to extend the language further, to be able to capture more complex adaptive behavior, e.g., multiple (possibly conflicting) goals, learning capabilities, and dynamically evolving actions and environments.

References

[1] Yuriy Brun et al. "Engineering Self-Adaptive Systems through Feedback Loops". In: *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*. 2009, pp. 48–70. DOI: 10.1007/978-3-642-02161-9_3. URL: https://doi.org/10.1007/978-3-642-02161-9%5C_3.

[2] Betty H. C. Cheng et al. "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Ed. by Betty H. C. Cheng et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 1–26. ISBN: 978-3-642-02161-9. DOI: 10.1007/978-3-642-02161-9_1. URL: https://doi.org/10.1007/978-3-642-02161-9_1.

[3] John H. Holland. "Studying Complex Adaptive Systems". In: *J. Systems Science & Complexity* 19.1 (2006), pp. 1–8. DOI: 10.1007/s11424-006-0001-z. URL: https://doi.org/10.1007/s11424-006-0001-z.

[4] Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa. "Integrating deployment architectures and resource consumption in timed object-oriented models". In: *Journal of Logical and Algebraic Methods in Programming* 1 (2015). DOI: 10.1016/j.jlamp.2014.07.001.

[5] *Sirius-Center*. http://sirius-labs.no/. Accessed: 2018-08-29.

Railway capacity verification as bounded model checking

Bjørnar Luteberget

University of Oslo / Railcomplete AS,
bjornalu@ifi.uio.no

This is a presentation abstract for the 30th Nordic Workshop on Programming Theory based on work done in collaboration with Claus Feyling (Railcomplete AS), Christian Johansen and Martin Steffen (Univ. Oslo), and Koen Claessen (Chalmers Univ.). Parts of this text were originally published in [1].

1 Introduction

Railway capacity is complex to define and analyze (see e.g. [2] or [3]), and existing tools and methods used in practice require comprehensive models of the railway network and its timetables. Design engineers working within the limited scope of construction projects report that only ad-hoc, experience-based methods of capacity analysis are available to them. Designs have subtle capacity pitfalls which are discovered too late, only when network-wide timetables are made – there is a mismatch between the scope of construction projects and the scope of capacity analysis, as currently practiced.

We consider one central problem that occurs when designing the layout and control systems for railway stations: Does the station infrastructure have the *capacity* to handle the amount of trains and the desired traveling times to provide adequate service in transportation of goods and passengers?

As an example, consider the question of crossing trains on a railway station. Figure 1 shows two sequences of movements which result in such a crossing. There are a number of details of the railway design which can cause this scenario to become infeasible (or take an unacceptably long time), such as signal placement, detector placement, correct allocation and freeing of resources, track lengths, train lengths, etc. More precisely: given a railway station track plan including signaling components, rolling stock dynamic characteristics, and a performance/capacity specification, verify whether the specification can be satisfied and find a dispatch plan as a witness to prove it.

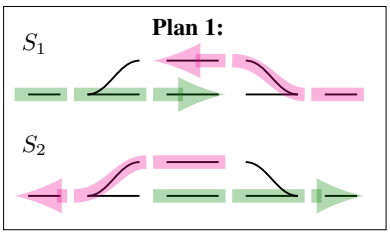


Figure 1: An example plan achieving a crossing of two trains on a two-track station. Green areas are currently occupied by a train going from left to right, while pink areas show are currently occupied by a train going from right to left.

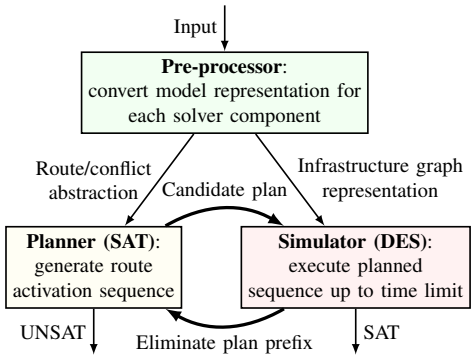


Figure 2: Conceptual diagram of solver CEGAR architecture. Inputs are transformed into (1) planner abstraction and (2) detailed simulation model.

2 Station performance requirements

To capture typical performance and capacity requirements in construction projects, we define an **operational scenario** $S = (V, M, C)$ as follows:

1. A set of **vehicle types** V, each defined by a length l, a maximum velocity $v_{\max}$, a maximum acceleration a, and a maximum braking retardation b.
2. A set of **movements** M, each defined by a vehicle type and an ordered sequence of visits. Each visit q is a set of alternative locations $\{l_i\}$ and an optional minimum dwelling time t_d.
3. A set of **timing constraints** C, which are two visits q_a, q_b, and and a maximum time difference.

As an example, we use the **crossing** scenario (Figure 1 shows a solution). Trains traveling in *opposite directions* can visit this station simultaneously[1].

```
movement passengertrain { visit #p_in [b1]; visit #p_out [b2] }
movement goodstrain { visit #g_in [b2]; visit #g_out [b1] }
timing p_in < g_out timing g_in < p_out
```

Similar specifications, and combinations of such specifications, are relevant in most railway construction projects. Since we typically only need to refer to locations such as model boundaries and loading/unloading locations, these specifications are not tied to a specific design, and can often be reused even when the design of the station changes drastically. Verification of these properties amounts to planning in a mixed discrete/continuous domain.

3 Separating dispatch planning from train dynamics

We have investigated logic-based approaches for the problem described above. The PDDL+ [4] language was designed to express planning problems in mixed discrete/continuous domains. Each discrete change needs a planning step, and our test case problem instances would need hundreds of steps to be solvable. We were only able to solve the most trivial test cases using the SMTPlan+ solver.

In response, we have developed a CEGAR-style tool which exploits the limited number of control system commands to make an abstraction of the planning problem, see Figure 2. This approach exploits the separation between the work that a dispatcher does (choosing which routes trains take) from the work that a train driver does (choosing when to start accelerating or braking based on available information about the path currently ahead). The dispatcher's planning can be solved by encoding the planning problem into SAT, see Figure 3, while the train movements and train driver behavior can be simulated using discrete event simulation [5], see Figure 4.

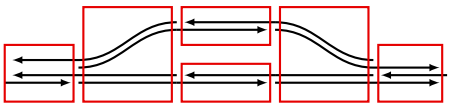


Figure 3: The planner component takes an abstracted view of the railway infrastructure. Lines represent elementary routes with traveling direction given by the arrows. Boxes indicate routes in conflict, i.e. only one of them can be in use at a time.

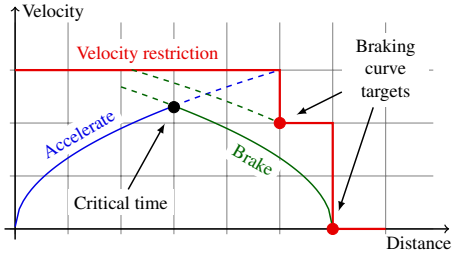


Figure 4: The train driver's decisions to accelerate/brake/coast happens at intersections between acceleration/braking/speed restriction curves.

[1] For details of the input file formats, see https://luteberget.github.io/rollingdocs/usage.html

4 Dispatch planning using SAT

We can encode the dispatch planning problem into an instance of the Boolean satisfiability problem (SAT). We consider the problem a model checking problem, and use the technique of bounded model checking (BMC) to unroll the transition relation of the system for a number of steps k, expressing state and transitions using propositional logic.

Using BMC for planning works by asserting the existence of a plan, so that when the corresponding SAT instance is satisfiable, it proves the fulfillment of the performance requirements and gives an example plan for it. When unsatisfiable, we are ensured that there is no plan within the number of steps k. In practice plans with higher number of steps are not of interest; i.e., the bound k is chosen based on practical considerations (twice the number of trains was sufficient in our case study). The SAT instance is built incrementally by solving with $k-1$ steps and then adding the k^{th} step if necessary.

A *state* i of the system in the planner component is represented by giving each route r_j an **occupancy status** $o^i_{r_j}$: it can be free ($o^i_{r_j} = \text{Free}$) or it can be occupied by a specific train t_k ($o^j_{r_i} = t_k$).

A dispatch plan is produced directly from the occupancy status $o^i_{r_j}$ of states by taking the difference between consecutive states and then dispatching any trains and routes which become active from one state to the next. Constraints on states ensure that (1) the plan is viable for execution (i.e., correctness), e.g. conflicting routes are not activated simultaneously (trains cannot collide), and (2) that the plan fulfills performance specifications.

5 Extensions

The SAT-based planning approach described above is part of a growing tool set for agile verification of railway designs, to aid the railway engineer while the design is still undergoing major revisions.

We are using the planner as a basis to extend in the following ways:

Simulation: To measure the time it takes to execute dispatch plans, one can use a discrete event simulation which represents the dynamic behavior of trains much more detailed than is tractable to model in an automated solver, see Figure 4. Commercial operational simulation packages such a OpenTrack [6] or LUKS [7] may be used for greater level of detail.

Optimization: The planner can be used to detect whether some equipment in the design is redundant. If a plan can be found which does require any use of certain pieces of signalling equipment, these pieces can be considered for removal from the design.

References

[1] B. Luteberget, K. Claessen, and C. Johansen, "Design-time railway capacity verification using SAT modulo discrete event simulation," in *FMCAD 2018 (to appear)*.

[2] M. Abril, F. Barber, L. Ingolotti, M. Salido, P. Tormos, and A. Lova, "An assessment of railway capacity," *Transportation Research Part E: Logistics and Transportation Review*, vol. 44, no. 5, pp. 774 – 806, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1366554507000579

[3] A. Landex, "Methods to estimate railway capacity and passenger delays," Ph.D. dissertation, 2008. [Online]. Available: http://orbit.dtu.dk/en/publications/id(f5578206-74c3-4c94-ba0d-43f7da82bf95).html

[4] M. Fox and D. Long, "Modelling mixed discrete-continuous domains for planning," *J. Artif. Intell. Res.*, vol. 27, pp. 235–297, 2006. [Online]. Available: https://doi.org/10.1613/jair.2044

[5] S. Robinson, *Simulation: The Practice of Model Development and Use*. USA: John Wiley & Sons, Inc., 2004.

[6] "OpenTrack: Simulation of railway networks," 2018. [Online]. Available: http://www.opentrack.ch/

[7] "LUKS: Analysis of lines and junctions," 2018. [Online]. Available: http://www.via-con.de/development/luks

On choreographies and communication failures

Fabrizio Montesi and Marco Peressotti

Concurrency and Logic Group, IMADA, University of Southern Denmark
fmontesi@imada.sdu.dk peressotti@imada.sdu.dk

Choreographies are high-level descriptions of communicating systems, inspired by the "Alice and Bob" notation for security protocols, where the behaviours of participants is defined from a global viewpoint. The hallmark of these descriptions is the communication primitive [7]. For example, the communication term `s.b -> r.f` reads "process `s` sends the bit `b` to process `r`, which handles the received message using the local function `f`". The idea is that correct local implementations of the processes described in a choreography (`s` and `r`) can then be automatically generated, yielding certified implementations.

The choreography models explored so far have the unrealistic assumption that communications never fail, which limits their correctness guarantees in the real world.

Solving this problem is challenging, because there is no "one size fits all" solution for handling failures in choreographies. For example, in choreographies for the Internet of Things, some communications between sensors and a collector might be expendable [4, 5], whereas in choreographies for parallel computing some communications must be performed to reach a correct result [1]. In general, depending on the underlying failure model and software requirements, the programmer might need to specify different recovery strategies for different communications in the same choreography. Further, the sender and receiver of a communication might have different local recovery strategies. For example, to implement the communication `s.b -> r.f`, we might want `s` and `r` to have different retry policies in case of failures: `s` would use exponential backoff between failed send attempts, while `r` would just wait for a fixed amount of time (timeout strategy).

In this talk we present Robust Choreographies (RC) a new programming model that brings choreographies all the way to being applicable to settings with realistic communication failures [6]. Differently from previous work on choreographies, all send and receive actions might fail in RC, modelling potential connection problems and/or timeouts on both ends.

The key novelty of RC is that the semantics of the choreographic communication primitive is programmable: communication is not offered as a primitive, but rather can be implemented using procedures. For example, we can define a parametric procedure `comAMO` that implements an at-most-once communication using an exponential backoff strategy for the sender and a timeout strategy for the receiver as follows.

```
1 def comAMO(s,f,r,g)
2   k:s -> r in              // declare a communication from s to r and name it k
3   sendExpBackoff(k,s,f);   // sender strategy for k
4   recvTimeout(k,r,g)       // receiver strategy for k
```

The procedure `comAMO` takes as formal parameters: the sender process `s` with its local function `f` to generate the payload, and the receiver process `r` with its local function `g` to handle the received payload. The communication that should take place is declared by the choreographic notation `k:s -> r` and named `k`. The implementations of the respective send and receive strategies for `k` are delegated to the procedures `sendExpBackoff` and `recvTimeout`. To implement our bit communication `s.b -> r.f` in RC as an at-most-once communication, we can simply invoke `comAMO(s,b,r,f)`.

RC has communication primitives to send and receive messages:

```
1 k <- s.f;  // s evaluates f and sends the result for k
2 r.g <- k;  // r receives the message for k and handles it with g
```

Communications are asynchronous and may fail. A successful send action implies that the sent message is now handed over to the communication stack of the sender, which will attempt at transmitting the message to the receiver. If transmission succeeds, the message reaches the receiver and is stored by the communication stack of the receiver in a dedicated memory. A successful receive action means that a message has been consumed by the intended receiver, *i.e.*, the message has been successfully *delivered*—this requires that transmission was successful. A receive action fails if it is executed when there is no message that it can consume. This models that there may be connection problems on the end of the receiver or that a timeout occurred on the receive action. We assume that communication and node failures are transient, meaning that failing to interact with a node does not impede eventually doing it in later retries. We leave persistent failures to future work. Processes can query their stack for the status of a message by means of the following guards:

```
3 if s.k! then C1 else C2 endif // chooses C1 if the last send by s for k succeeded
      and C2 otherwise
4 if r.k? then C1 else C2 endif // chooses C1 if the last receive by r for k succeeded
       and C2 otherwise
```

Aiming at providing a foundational choreography model, RC is designed around few simple primitives yet it is expressive enough to implement common recovery strategies. For instance, we can use the procedure `sendWhile` (and its dual for receiving) to implement `sendExpBackoff` (and `recvTimeout`) or any strategy that follows the same pattern: repeatedly attempt to communicate until successful or a certain condition is no longer verified.

```
1 def sendWhile(s, k:s->r, f, g, f')
2   if s.g ∧ ¬s.k! then // if s can try to send k
3     k <- s.f;          // s computes the payload with f and sends it
4     s.f';              // s updates its status by running f' (e.g. step a counter)
5     sendWhile(s,k,f,g,f');
6   endif
```

We can also implement more complex communication primitives, like those in [3, 5]. For instance, below is a procedure that iteratively attempt at sending some messages until the sender stack accepts all of them using a round-robin strategy.

```
1 def sendAll(s,k₁,f₁,...,kₙ,fₙ)
2   if s.k₁! then
3     sendAll(s,k₂,f₂,...,kₙ,fₙ) // omit this line to get sendAny
4   else
5     k₁ <- s.f₁;
6     sendAll(s,k₂,f₂,...,kₙ,fₙ,k₁,f₁)
7   endif
```

By omitting the recursive call at Line 3 we obtain a similar procedure that instead stops as soon as any of the messages is sent.

The evocative notation for expressing communications typical of choreographies is readily recovered by means of some syntax sugar; below are some examples.

```
1 k:s.f ->_t r.g  // each party gives up after timeout t for t > 0.
2 k:s.f ->^n r.g  // each party gives up after n attempts for n > 0
3 k:s.f ->* r.g  // each party makes an unbounded number of attempts.
```

Many choreography models can be compiled to RC. For instance, we can recover the language of Procedural Choreographies (PC) [2], which abstracts from communication failures, "as a library". Translating PC programs into RC is a matter of rewriting a few symbols, *e.g.*, `s.f -> r.g` in PC becomes `s.f ->* r.g` in RC.

In this talk:

- We present RC, formalise its semantics, and show how it captures different failure models thus allowing programmers to deal with different kinds of systems such as: local Inter-Process Communication (IPC) mechanisms (like unnamed pipes in POSIX systems, shared memory, or file-based communications), reliable message delivery protocols (like TCP, under the assumption that there are no connection resets or similar issues), unreliable message delivery protocols (like UDP).
- We illustrate the programming model and expressiveness of RC by implementing common recovery strategies and distributed protocols.
- We introduce a simple type system that checks that communications have consistent implementations (parties are connected, payload types are respected, and communication attempts are matched) and prove that well-typed programs enjoy progress, *i.e.*, they either terminate or diverge.
- We define a formal translation (a compiler, if you like) from choreographies in RC to a more standard process model, *i.e.*, an asynchronous variant of the π-calculus equipped with standard I/O actions that might fail. These asynchronous fallible I/O actions are the only way processes may interact: there is no shared memory or agreement primitive. We prove that, if the original choreography is well-typed, the synthesised code is operationally equivalent and enjoys progress.

References

[1] L. Cruz-Filipe and F. Montesi. Choreographies in practice. In *FORTE*, volume 9688 of *LNCS*, pages 114–123. Springer, 2016.

[2] L. Cruz-Filipe and F. Montesi. Procedural choreographic programming. In *FORTE*, LNCS. Springer, 2017.

[3] L. Cruz-Filipe, F. Montesi, and M. Peressotti. Communication in choreographies, revisited. In *SAC*. ACM, 2018. To Appear.

[4] H. A. López and K. Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In *SAC*, pages 437–443. ACM, 2017.

[5] H. A. López, F. Nielson, and H. R. Nielson. Enforcing availability in failure-aware communicating systems. In *FORTE*, volume 9688 of *Lecture Notes in Computer Science*, pages 195–211. Springer, 2016.

[6] F. Montesi and M. Peressotti. Choreographies meet communication failures. *CoRR*, abs/1712.05465, 2017.

[7] W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.

A Language for Modelling Privacy

Ian Oliver

Nokia Bell Labs
Espoo, Finland
ian.oliver at nokia-bell-labs.com

Abstract

A language for modelling information flow from a privacy perspective is required in information system engineering. We present such a language based on a data flow model with annotations and rules linking privacy ontologies with said model. A graphical syntax with a naïve and informal semantics has sufficed in current usage but lacks a strong semantic basis for successful and meaningful integration with programming languages and other development tools. The need for the semantics and an outline of the structuring is outlined.

1 Introduction

Privacy is a hot topic and while amply catered for regarding legal aspects, privacy as an aspect of the information system engineering process and artefacts is little more than a collection of various methodologies and processes [2, 3]. We introduce a small, graphical language for the expression of the flow of data through an information system coupled with an ontology of concepts for the expression of information type, usage, purpose and provenance, as well as rules for the inference of privacy issues and relationship to a requirements model.

There have been relatively few attempts at languages which explicitly address privacy and information flow at either a modelling or programming level. One has been Jeeves [10] which addresses the explicit inclusion of policies to guide the results of queries and functions. However Jeeves is (by design) too low-level for the system modelling and possible simulation of information flows and concentrates on a policy based approach which has a number of issues when applied at a higher-level of abstraction [4]. Another attempt has been to provide a modelling language with an MDA-style model transformation in [1] similar to that earlier made in [6].

The language presented here was developed with both the system engineer/architect and privacy legal in mind through a set of concepts backed by ontologies to allow both communities to reason about the information flow, system architecture and privacy.

This paper briefly introduces the concepts and provides a basis for a formal semantics and is aimed to open up discussion on what formalism is most suitable for such a heterogenous language, either in the modelling or programming domain. The full graphical language and associated privacy ontologies are described in [6] and only a very short extract is presented below to provide some structure to the semantics.

2 Syntax and Towards a [Formal] Semantics

Figure 1 shows a simple example denoting the flow of data via some camera application to some storage. The data is collected from what the camera 'sees' on behalf of a data subject[1].

Information flows are annotated by their contents using various ontologies [5] that attempts to bridge the gap between programming language structures and types and the ubiquitous 'type' *personal data*. The top-level information class is shown in figure 2 and an example of a diagram annotated by usage classifications in 3.

[1] A data subject as defined by the EU GDPR is any person whose personal data is being collected, held or processed

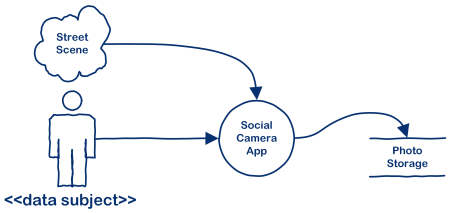


Figure 1: Example Data Flow

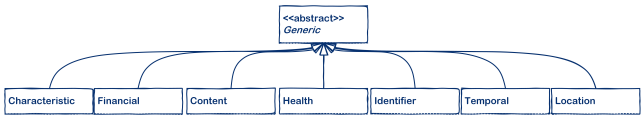


Figure 2: Top Level Information Type Classes

A model is defined as a set of nodes, flows and functions for extracting the source and target of flows and annotations of any element ($Nodes \cup Flows$), and a usual set of rules, eg: $\forall f : Flows, src(f)! = targ(f)$ which is generalised as the flows are acyclic [8] over the model[2].

$$Model : (Nodes, Flows, src, targ, annotations)$$

Annotations refer to the structured labelling of elements, for example, the usages of the data in the storage elements (fig.3), or annotation of data flow content (fig.1). Annotations are structured according to a set of ontologies (Data Type, Usage, Provenance, Purpose etc) - these ontologies are defined using description logic but limited, generally, to $\mathcal{ALC}(\mathcal{F})\mathcal{H}$. These constitute the first set of rules over the model.

We can define the semantics of some processes as a second set of rules which define the behaviours of certain elements under certain conditions. For example, for a process node ($Process \sqsubseteq Node$), the outputs must refine the inputs:

$$\begin{aligned} &\forall n : Process \\ &\quad \text{let} \quad inp = \{f : Flows \bullet \text{targ}(f) = n | annotation_{dataType}(f)\} \\ &\quad \text{let} \quad out = \{f : Flows \bullet \text{src}(f) = n | annotation_{dataType}(f)\} \\ &\quad out \sqsubseteq inp \end{aligned}$$

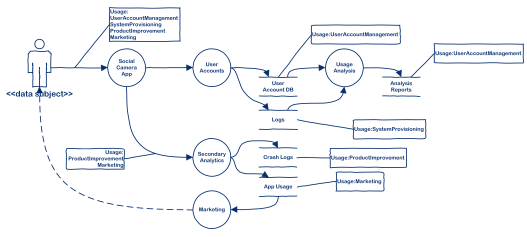


Figure 3: Example Usage Annotation in a DFD

[2]In the full syntax we have a specific type of 'reverse' flow for this explicit occurrence, eg: marketing flows back to the data subject.

A third class of rules of the model is given by the behaviour of certain ontologies. For example, usages of data must (should!) decrease through the model:

$$\forall n : Process \\ \cup \{f : Flows \bullet \text{targ}(f) = n | annotation_{usage}(f)\} \subseteq \\ \cup \{f : Flows \bullet \text{src}(f) = n | annotation_{usage}(f)\}$$

A fourth class of rules deal with certain ontological conditions such as equivalences and implications between certain ontological elements. A canonical example is taken from the data type ontology where certain machine identifiers can be geolocated, for example, IP addresses can be equivalent Locations in types of type with some interesting effects upon the system.

There are other constructs such as the logical partitioning of models which we have not discussed here but have particular meaning in the analysis of the model.

Finally under analysis the models are mapped to further constructs - similarly defined using ontologies - for requirements management and risk analysis [9]. These are discussed in more detail in [7].

3 Conclusions and Future Work

We have introduced a small, graphical and informally defined language for the expression of information flows with respect to privacy. We have also show the link between the data flow and ontological structures for privacy as well as their mapping to requirements and risk structures.

While having some structure to the rules of the model we lack a full and proper formal semantics. While for most industrial applications the informal rules suffice, it does complicate the implementation of tools supporting this modelling language.

References

[1] Thibaud Antignac, Riccardo Scandariato, and Gerardo Schneider. Privacy compliance via model transformations. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, April 2018.

[2] Jose M. del Alamo, Yod-Samuel Martín, and Julio C. Caiza. *Towards Organizing the Growing Knowledge on Privacy Engineering*, pages 15–24. Springer International Publishing, Cham, 2018.

[3] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering*, 16(1):3–32, Mar 2011.

[4] Sabrina Kirrane, Serena Villata, and Mathieu d'Aquin. Privacy, security and policies: A review of problems and solutions with semantic web technologies. *Semantic Web*, 9(2):153–161, 2018.

[5] Annie I. Anton Ninghui Li, Ting Yu. A semantics-based approach to privacy languages. Technical report, CERIAS.

[6] Dr Ian Oliver. *Privacy Engineering: A Dataflow and Ontological Approach*. CreateSpace Independent Publishing Platform, USA, 1st edition, 2014.

[7] I. Oliver. Experiences in the development and usage of a privacy requirements framework. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 293–302, Sept 2016.

[8] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[9] Stuart S. Shapiro. Privacy risk analysis based on system control structures: Adapting system-theoretic process analysis for privacy engineering. In *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016*, pages 17–24. IEEE Computer Society, 2016.

[10] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. pages 85–96, 2012.

Towards Semantical Foundations of Services Computing

Tobias Rosenberger[1], Saddek Bensalem[2], Marius Bozga[2], and Markus Roggenbach[3]

[1] Swansea University, Université Grenoble Alpes
tobias.rosenberger@univ-grenoble-alpes.fr
[2] Université Grenoble Alpes
{saddek.bensalem,marius.bozga}@univ-grenoble-alpes.fr
[3] Swansea University
m.roggenbach@swansea.ac.uk

Software services are ubiquitous. They are accessed from PCs, mobile devices, car cockpits, control panels in production plants, etc. They control the physical actions of machines and money transfers; they handle personal information [GK17, Uni] and support decisions and activities, be they political, professional or private. They aggregate news and social media posts, recommend books and holiday destinations. We routinely place life, liberty and property in their hands. The work presented here is part of an effort [Ros18a, Ros18b] to ensure that services are worthy of that trust: we want to verify, as far as possible, the preservation of safety and security properties under composition of software services.

The main contribution we present here is a transformational semantics of *gRPC* [Goo][1] in *BIP* [BBB+11]. Source to source transformations are a common part of a *BIP* based development process [BBB+11]. *gRPC* is a remote procedure call technology and interface definition language used for connecting software services using a standardised binary on-the wire format with support for backwards-compatible changes and for various programming languages. *gRPC* is used internally by Google to connect their various services [The], and has also been adopted by other organisations like Netflix [Net] (to replace their internal RPC solution), Cisco [Cis] (to configure BGP routers, which are used for routing between Internet Service Providers) and Cockroach Labs [The] (for their open source cloud based distributed SQL server).

Let us now look at a simple example of a *gRPC* interface definition:

```
1 syntax = "proto2";
2 message Int {required uint32 content = 1;}
3 service Echo {rpc echo (Int) returns (Int);}
```

It specifies first a message type *Int*, which here just contains one *required* unsigned 32 bit integer (*uint32*) entry called *content*. The 1 in the message entry is a numeric identifier the same way *content* is a textual one and is used to encode data in the binary on-the-wire format of *gRPC*. In general a message could be a record with *optional*, *required* and *repeated* entries, with groups of alternatives and with nesting of messages.

The *gRPC* code then goes on to specify a service *Echo* offering a single remote procedure call (*rpc*) called *echo* both takes and returns a message of the type *Int* we just defined.

BIP (Behavior-interaction-priority) [BBB+11] is a framework for developing and verifying component based systems with a strong separation of concerns between communication patterns, the internal behaviour of components participating in the communication patterns, and scheduling priorities. It specifies components, multi-party communication patterns and priorities in a Domain Specific Language and can bind to functions and types defined in general purpose programming languages, typically C++.

[1] Here we always use *gRPC* in the standard way with protocol buffers, and treat the two as a single concept. *gRPC* documents mostly assume the use of protocol buffers.

We will now show excerpts from a translation of the above *gRPC* echo server into *BIP*, the purpose being to give the reader a first impression of the *BIP* language and a hint at the principles of our translation, not to give a complete understanding of the resulting components. The *BIP* concepts we will present are atomic components with control state and data, and ports, which are exposed by components and to which in a composite system *BIP*'s connectors would attach. The reader will notice that the *BIP* translation is much longer than the *gRPC* specification, since it has to spell out properties that in *gRPC* are common to all services and can therefore be left implicit; the *BIP* version could be considered more low level.

```
4  extern data type uint32

7  extern function bool doneEcho(EchoState state)
8  extern function bool failedEcho(EchoState state)
9  extern function readEchoState(uint32 content, EchoState state)
10 extern function writeEchoState(EchoState state, uint32 content)
11 extern function uint32 readEchoError(EchoState state)
```

We bind to some C++ types and and functions we will use in the atomic components; below we will show excerpts of one of the components.

```
25 port type Echo_Comp_Accept(uint32 client, string_map client_metadata,
       uint32 deadline, uint32 content)
```

We define a port type for transferring the argument field *content*, as well as associated meta-information for the remote procedure call. The port type will later be instantiated as a communication port of a component. In reality we would usually have several fields to transfer.

```
29 atom type Echo_Comp()
30   data uint32 client
31   data string_map client_metadata
```

We see the beginning of an atomic component type containing some data fields. *BIP* has atomic and composite components.

```
40     export port Echo_Comp_Accept accept(client, client_metadata, deadline,
           arg_content)

44     place READY, COMPUTING
45     initial to READY
46     on accept from READY to COMPUTING do {
47       writeEchoState(state, arg_content);
48       done = false; failed = false;}
```

We see more details of the component: it contains a port of the type defined above. We see an atomic component is essentially a Petri net – in this simple case a state machine – with a finite number of places (states), with an initial transition and with transitions triggered by communicating over a port (*on accept ...*). There could also be purely internal transitions.

We associate with the transition an action calling an external function and setting two variables. The external function here is meant to initialise the *state* of the internal variables from the argument (field *arg_content*) which should have been set over the exposed port.

The *BIP* system naturally lends itself to our goals. It generates actual deployable software. It allows for incremental development and model-checking of component based systems, i.e., the system tries to avoid re-checking properties that can't be affected by changes [BBB+11]. The *BIP* ecosystem also includes tools for transforming component based systems into more efficient monolithic systems while maintaining the advantages of modular development and model-checking [BJS10], for model-checking of observed efficiency properties on a specific platform [NBB+15] and for modelling dynamic architecture reconfiguration [BBBS18]. All of these

features are of particular interest in the world of software services, where change between different atomic or distributed architectures or change of service providers based on Quality of Service properties (e.g., response time, scalability, availability) are among the main advantages of the Services Computing approach.

We give a simple correctness argument. To do so, we first develop formal semantics for a subset of *gRPC* in terms of simple automata, based on the official informal specification [Goo]. As is usual, transition from the formal to the informal is the crucial step where both the correctness and usefulness of our approach need to be examined. Seeing the equivalence between the automata and the generated *BIP* code, both in terms of data and the structure of the underlying transition systems, is then straightforward. We will further discuss an example of how security properties can be checked in our setting.

This semantics-preserving translation and our verification example are first steps to a verification framework for Services Computing which we want to develop within the *BIP* ecosystem. Since *gRPC* does not specify the actual computations, but only interfaces and communication patterns, the results of the translation are general *BIP* components which need further refinement into the actual implementation. That implementation can then be compiled into a service communicating via *gRPC* with other services developed with or without our framework. The framework should be easily extensible to interface definition languages and communication technologies other than *gRPC*, by giving semantics-preserving translations analogous to ours.

References

[BBB+11] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous component-based system design using the BIP framework. *IEEE software*, 28(3):41–48, 2011.

[BBBS18] Rim El Ballouli, Saddek Bensalem, Marius Bozga, and Joseph Sifakis. DR-BIP - programming dynamic reconfigurable systems. Technical Report TR-2018-3, Verimag Research Report, 2018.

[BJS10] Marius Bozga, Mohamad Jaber, and Joseph Sifakis. Source-to-source architecture transformation for performance optimization in BIP. *IEEE Transactions on Industrial Informatics*, 6(4):708–718, 2010.

[Cis] CiscoDevNet. https://github.com/CiscoDevNet/grpc-getting-started. accessed 2018-08-23.

[GK17] Aditi Gupta and Rishabh Kaushal. Towards detecting fake user accounts in facebook. In *Asia Security and Privacy (ISEASP), 2017 ISEA*, pages 1–6. IEEE, 2017.

[Goo] Google Inc. gRPC concepts. https://grpc.io/docs/guides/concepts.html. accessed 2018-06-01.

[NBB+15] Ayoub Nouri, Saddek Bensalem, Marius Bozga, Benoit Delahaye, Cyrille Jegourel, and Axel Legay. Statistical model checking QoS properties of systems with SBIP. *International Journal on Software Tools for Technology Transfer*, 17(2):171–185, Apr 2015.

[Net] Netflix. https://github.com/Netflix/ribbon. accessed 2018-08-23.

[Ros18a] Tobias Rosenberger. Verified safe and secure software services, March 2018. Talk at British Colloquium for Theoretical Computer Science 2018, Royal Holloway, University of London.

[Ros18b] Tobias Rosenberger. Verified safe and secure software services, June 2018. Poster Session at the College of Science Postgraduate Research Student Conference 2018, Swansea University.

[The] The gRPC Authors. https://grpc.io/about/. accessed 2018-08-23.

[Uni] United States Department of Health and Human Services. https://www.healthdata.gov/content/data-api. accessed 2018-06-01.

An Event-B Model for a Basic Prototype of a Healthcare System

Luigia Petre[2,3]*, Usman Sanwal[1,3]*, Gohar Shah[1,3], Charmi Panchal[1,3], Dwitiya Tiwari[1,3], and Ion Petre[1,3]

[1] Computational Biomodeling Laboratory
[2] Distributed Systems Laboratory
[3] Åbo Akademi University and Turku Centre for Computer Science
Turku 20500 Finland

Abstract

How robust is a healthcare system? How does a patient navigate the system and what is the cost (money, time, loss of productivity, decrease in quality of life) incurred from the first symptoms to getting cured? How will it fare in the wake to a sudden epidemic or a disaster? How are all of these affected by administrative decisions such as allocating/diminishing resources in various areas or centralising services? These are the questions motivating our study on a formal prototype model for a healthcare system. We propose that a healthcare system can be understood as a distributed system with independent nodes (healthcare providers) computing according to their own resources and constraints, with tasks (patient needs) being allocated in queues between the nodes. We construct in this paper an Event-B model capturing the basic functionality of a simplified healthcare system: patients with different types of medical needs being allocated to suitable medical providers, waiting for their turn for multi-step treatments depending on their needs.

1 Introduction

Healthcare systems are highly complex environments involving many different stakeholders (e.g., clinicians, patients, administrators) with highly diverse objectives (e.g., driven by medical concerns, need of effectiveness, need of efficiency, focus on costs or on service availability). Changes in a healthcare system are almost always driven by economical or administrative constraints and it is highly difficult to predict their consequences on the overall patient-focused quality of the system. We are interested in this paper in describing some of this complexity by building a formal model capturing the basic architecture of a healthcare system (medical providers with specific capabilities, and the connections between them) and the way patients navigate the system to have their medical needs met. The model captures the connections between the users and their first point of contact (local clinics), and the connections between the providers, from the small local units to the highly-specialized units, with the aim of capturing the path of a user through the system from general nurse advise to sophisticated treatments by specialist teams. The user is assumed to have registered at the nearest primary provider. Depending on the needs of the user, the primary provider suggests the preferred secondary provider where the required service could be obtained. A user can have multiple needs and a provider can be connected to many other providers. At each provider the patients are allocated in a waiting queue, which advances according to the capacity of that provider. A patient will stay longer in the queue for more difficult medical problems, with her needs being addressed

*Authors with equal contribution.
*Authors with equal contribution.

step-by-step until solved. In this prototype model we only include three generic categories of medical needs, differentiated through the treatment iterations that a patient must go through depending on her type of medical need.

The model we build in this paper uses the state-based Event-B formalism [2]. Event-B is an extension of the B-method [1], with elements of Action Systems [3], TLA [5], and UNITY [4], introduced for modelling and reasoning about systems and software. All Event-B models discussed in the paper are available at `http://users.abo.fi/ipetre/health-eventb-model.zip`.

2 Refinement-based construction of the Event-B model

The overall structure of our healthcare model is illustrated in Figure 1. We developed the model in three layers, represented by the machines *M0*, *M1*, and *M2*. These layers are linked by the refinement relation: $M0 \sqsubseteq M1 \sqsubseteq M2$. Machine *Mi* sees the context *Ci*, for all $i \in \{1,2,3\}$. The three contexts are linked by the extension relation, so that *C0* is extended by *C1*, which is then extended by *C2*.

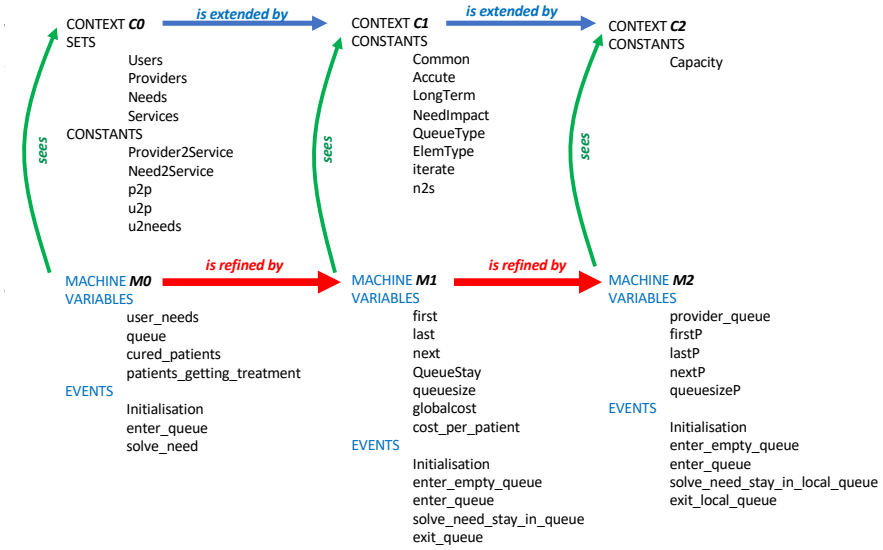


Figure 1: Overview of the model development.

Our basic model *M0* is concerned with defining the main elements of our healthcare system prototype: *users*, which may have various medical *needs*, and healthcare *providers* which provide different types of *services* to the admitted patients. These are defined in context *C0* as shown in Figure 1. The constants in *C0* are used to model the different facilities provided by the healthcare unit. For example each healthcare provider has a list of specific services that they can offer. This is modelled in *M0* through constant $Provider2Service$. The other services such as medical need, a primary provider for each user, relationship between user to provider and provider to provider are also modelled in *M0* with the help of constants defined in *C0*.

Model *M0* has 2 other events in addition to initialisation: *enter_queue* and *solve_need*. Event *enter_queue* models the insertion of a user-needs-provider tuple into the healthcare system, to be treated while the event *solve_queue* selects a non-empty subset *patients* from *queue*, removes them from the *queue* and adds them into the set of cured patients. Thus in the

model *M0* we define the sets and main variables that appear in our system, allocate patients in the system queue based on the ability of providers to treat them and treat patients non-deterministically. Even though very abstract, we model the fact that patients that enter the system are either under treatment (in *queue*) or cured (in *cured_patients*).

In the second model *M1* we have two aims. First, we model the queue of patients as a first-come-first-served structure, rather than as a set as was done in *M0*. Second, their stay in the system depends also on the type of medical need they have. We store the waiting time per patient, as well as the global waiting time of the entire queue.

Machine *M0* is refined to machine *M1* via a superposition refienement: we add seven new variables in this machine and refine the events in *M0* to assign appropriate values to these new variables. We also have new events in *M1* such as event *solve_need_stay_in_queue* which models one treatment step and event *exit_queue* which selects an arbitrary set of patients in the top of the queue that are cured, and takes them out of the queue, by updating accordingly all the relevant variables.

We introduce in the model *M2* the capacity of each provider. Patients are considered in the providers' own queues and they are treated only when these providers have resources (e.g., available doctors). This is the object of our third refinement.

Event-B turned out very useful in ensuring the consistency of the models. Model *M0* had 13 proof obligations (of which 12 automatically discharged), while *M1* had 53 (of which 35 were automatically discharged), while *M2* had 55 (of which 37 were automatically discharged).

3 Discussion

We constructed in this paper a prototype, EventB-based formal model of a healthcare system consisting of health service providers of various types, and patients with various types of needs. The main motivation in building such a model was to be able to reason formally about global measures of a healthcare system's quality and robustness, such as average waiting times under "normal" and under "stress" system loads, average duration from disease onset to curation for disease of different intensities. We also aimed to be able to measure the effect of various types of service cuts or service reorganisations on these measures of quality/robustness. The work so far covered the construction of a prototype model of a simplified healthcare system. The work is now in progress on using this model towards measures of quality and robustness in this model.

References

[1] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] Ralph-Johan Back and Reino Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, pages 131–142, New York, NY, USA, 1983. ACM.

[4] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[5] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Soft- ware Engineers.* Addison-Wesley, 2003.

A Roadmap for Multi-Model Consistency Management

Patrick Stünkel[1], Harald König[2], Yngve Lamo[1], and Adrian Rutle[1]

[1] Western Norway University of Applied Sciences, Bergen, Norway
{past,yla,aru}@hvl.no
[2] FHDW Hannover, Hanover, Germany
Harald.Koenig@fhdw.de

1 The Global Consistency Challenge

Contemporary software development does not simply comprise writing a single program but it usually involves a multitude of interrelated and heterogeneous artifacts: source code files, requirements specifications, database schema, external interface specifications, program libraries, etc. *Consistency Management* among these artifacts is arguably the most crucial activity of the software development process. Furthermore, it is also crucial for the operation of software systems, i.e. multiple systems have to work together and exchange information consistently. Tools and frameworks for *local* consistency checking, e.g. syntax checkers, database integrity constraints, form validators, etc., are widely established and used. However, consistency is not limited to a single artifact but has to be maintained *globally*, e.g., use cases from a requirements specification should correspond to implemented functionality of the program, entities in a class diagram should correspond to database tables, etc. The whole process gets even more challenging as the artifacts are usually distributed over organizational and technical domains.

Global consistency maintenance is hence a key challenge that has to be addressed in Software Engineering to fulfil expectations in major trends like eHealth, Industry 4.0 automation and cyber-physical systems. Furthermore, it is one of the key factors for the success of Model Driven Software Engineering (MDSE) [8].

2 Conceptual Framework

First, one needs a vision of a conceptual framework that captures all involved artifacts of the software development process in which consistency maintenance is performed. In the spirit of MDSE, these artifacts will be called *models*, i.e abstract representations of software artifacts. A model can be an executable piece of program code, a colloquial system requirements description, a class diagram and many more.

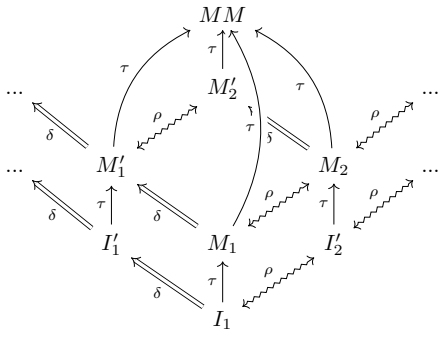


Figure 1: Global Consistency Maintenance Framework

Next, there are relationships between models. One important relationship, which has extensively been studied in the MDSE community [2] is *conformance* (denoted by τ in Fig. 1): A model (called metamodel) gives rise to a model language; i.e., a class of models (called instances) that adhere to rules imposed by the metamodel. A metamodel is associated with an environment or system: e.g., a programming language specification defines the class of all programs written in this language. The other important relationship we call for *correspondence* (denoted by ρ

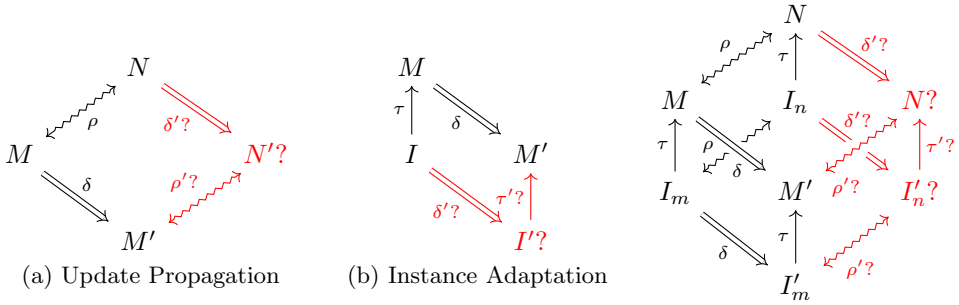
in Fig. 1). It occurs when multiple models share the *same* concepts; i.e., describing some real world phenomena from different views. For example, a real world entity may be represented in a class diagram as well as a table in database scheme. Finally, all models are subject to change (denoted by δ in Fig. 1), which may happen due to new requirements, changed laws and regulations, etc. This represents the third relationship type between models in this framework.

Together these concepts form the notion of a *Global Consistency Maintenance* framework. In figure 1, an example of such a framework containing models on three "levels", e.g. metamodels (MM), models ($M_1, M_2, ...$) and instances ($I_1, I_2, ...$) having binary correspondence relations among them is sketched. However, a general framework is not limited to three metalevels and binary correspondence relations; i.e., it might be multi-level and/or include multi-ary relations. Moreover, the main challenge is to maintain consistency w.r.t. changes; i.e., (a) *checking* conformance and correspondence relations, and (b) *restoring* them if necessary in the demeanor of changes. In this work we propose solutions to address this challenge. However, the support of the full-fledged setting (2c) is still subject to future work.

3 Consistency Maintenance

Consistency Maintenance can be divided into two subtasks: (i) *Consistency checking*, which can in turn be divided into conformance checking (do models adhere to syntactical and semantical rules of the metamodel?) and correspondence checking (are models free of contradictions?), and (ii) *Consistency restoration*, which can in turn be divided into various scenarios as depicted in fig. 2. The scenario *Update Propagation* (2a) for the case of binary ρ has been well treated by the cross-disciplinary research field *bidirectional transformations (bx)* [1, 7]. The general multi-ary situation only recently came into focus and is still an open field [8]. The scenario *Instance Adaptation* (2b) has been treated in databases under the topic *schema/instance adaptation* [4] and in the MDSE community as *metamodel/model co-evolution* [5]. So far, to our knowledge, there are no approaches that deal with conformance and correspondence at the same time (2c) in a setting that comprises conformance relations between multiple levels and general multi-ary correspondence relations.

This work is associated with the authors work in [9], which represents a first step towards this general framework: It is based on the *Diagram Predicate Framework (DPF)* [6], a formalism for MDSE. DPF is capable of representing models (by means of graphs) on arbitrary levels and formalizes the conformance relation by *typing* and *satisfaction of diagrammatic constraints*. In [9] a notion of correspondence is added, which is expressed by multi-ary graph spans, i.e. an auxiliary model R is introduced that connects related models by graph morphisms. Consistency of the correspondence relation is expressed by diagrammatic constraints imposed on a *merged view* over this relation. The resulting framework is capable of consistency checking but is



(a) Update Propagation (b) Instance Adaptation (c) Combined Notion

Figure 2: Multi Modeling Situations

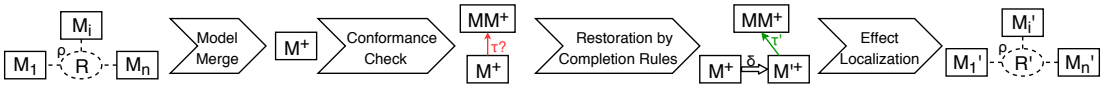


Figure 3: Consistency Restoration Procedure

lacking a consistency restoration mechanism. Thus, the next stage of our work is to establish such a mechanism. The idea is to exploit *Completion Rules* [3], i.e. equipping constraints with transformation rules that repair inconsistent instances. Completion Rules can be classified as an Instance Adaptation approach.

Figure 3 depicts the proposed consistency restoration procedure: Let $M_1, , \ldots, M_n$ be a collection of models that are in a correspondence relation, expressed by a multi-span R. To check consistency by means of the method introduced in [9], one first has to construct the merged view M^+ over the correspondence by a colimit construction. The resulting comprehensive merged view M^+ is then checked against a comprehensive metamodel MM^+ that comprises *inter-model constraints* which tell whether the correspondence can be considered as consistent. Such a comprehensive metamodel MM^+ could in turn be constructed by calculating the merge of respective metamodels $MM_1, \ldots, MM_n$. If M^+ does not conform to MM^+, completion rules are applied to yield an updated M'^+. Altogether a local change (e.g. on model M_1), which caused global inconsistency, is repaired in the virtual merge M^+ and the effect is then propagated back into all other local models $M_2, \ldots, M_n$ (localization). Our current research focuses on this restoration procedure, under which criteria it is applicable, its formal foundations given by categorical universal constructions, restrictions on the type of supported models (e.g. data models, behavioral specifications, etc.), and its relationship to existing consistency restoration frameworks.

References

[1] Soichiro Hidaka, Massimo Tisi, Jordi Cabot, and Zhenjiang Hu. Feature-based Classification of Bidirectional Transformation Approaches. *Softw. Syst. Model.*, 15(3):907–928, jul 2016.

[2] Thomas Kühne. Matters of (Meta-) Modeling. *Software & Systems Modeling*, 5(4):369–385, December 2006.

[3] Fazle Rabbi, Yngve Lamo, Ingrid Chieh Yu, and Lars Kristensen. A Diagrammatic Approach to Model Completion. In *AMT@MODELS '15*, pages 56–65, 2015.

[4] John F. Roddick. Schema Evolution in Database Systems: An Annotated Bibliography. *SIGMOD Rec.*, 21(4):35–40, December 1992.

[5] Louis M. Rose, Markus Herrmannsdoerfer, Steffen Mazanek, Pieter Van Gorp, Sebastian Buchwald, Tassilo Horn, Elina Kalnina, Andreas Koch, Kevin Lano, Bernhard Schätz, and Manuel Wimmer. Graph and model transformation tools for model migration. *Software & Systems Modeling*, 13(1):323–359, February 2014.

[6] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, University of Bergen, 2010.

[7] Perdita Stevens. A Landscape of Bidirectional Model Transformations. In *Generative and Transformational Techniques in Software Engineering II*, Lecture Notes in Computer Science, pages 408–424. Springer, Berlin, Heidelberg, July 2007.

[8] Perdita Stevens. Bidirectional Transformations In The Large. In *MODELS 2017*, pages 1–11, June 2017.

[9] Patrick Stünkel, Harald König, Yngve Lamo, and Adrian Rutle. Multimodel correspondence through inter-model constraints. In *"Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming(Companion)"*. ACM, 2 2018.

Language-Based Support for GDPR-Related Privacy Requirements

Shukun Tokas and Toktam Ramezanifarkhani

Department of Informatics, University of Oslo, Norway

Introduction

With the adoption of information technology in almost all areas of our life, the collection and processing of personal data have intensified. This development depends on the trustworthy functioning of information and communication technologies (ICT) to support individual's personal rights and democratic values of society [1]. To address the challenges of data protection and privacy of individuals within European Union (EU) and European Economic Area (EEU), the European Union Parliament approved the General Data Protection Regulation (GDPR) [2]. The major focus of the GDPR is on the *consent* and *purpose* of collection, processing, and sharing of the personal data. GDPR also restricts who (*role*) can access this personal data, allowing access to only the authorized "principals" (i.e., the controllers and the processors). Moreover, the GDPR regulations (Article 25 [2]) call for means to establish *privacy by design*, i.e., identifying the privacy requirements early at the design stage and embedding them into the subsequent implementation that handles the personal data.

In this paper, we suggest a methodology for privacy by design, which integrates necessary safeguards into the information processing by using language-based mechanisms. Particularly, we enforce *memory safety* using a type-system based approach. GDPR compliance is defined in natural language, and is therefore difficult to check in practice for a given program. Certain aspects of GDPR can be expressed by means of static concepts, while others can only be expressed at run-time, such as user-defined changes in privacy restrictions. Our ambition here is to capture a meaningful static GDPR notion, by formulating policies that can be checked statically. We consider static concepts such as method names, sets of methods names, interfaces, co-interfaces (as explained below) and access rights given by *read/write/increment*. In particular, we focus on the concepts of *role* and *purpose*, as they express key aspects of GDPR specific privacy policies. We do not formalize the concept of consent, since changes in consent are dynamic by nature. However, consent can implicitly be seen as the presence of a policy.

To formalize our approach we use a high-level kernel language for object-oriented, distributed systems, inspired by the *Creol* [3] concurrency model, because it has a modular semantics and allows us to focus on the information flow at high level of abstraction. The Creol language has earlier been extended with basic secrecy level (high and low) [4], but not privacy notions. We give policies for private data as well as objects encapsulating private information. A policy will put restrictions on what kind of principals may access the private information, also on why (for what purpose) the information can be accessed, and what kind of operations are allowed on this data i.e., restricting *who*, *why* and *what*. The *who*, *why* and *what* restrictions correspond to *role*, *purpose* and *access*. For a data value with private information, the associated policy is given by a set of triples $(role; purpose; access)$ where *role* is given by an interface, *purpose* is given by a set of functions/methods (or alternatively *any*, denoting any purpose), and access restriction is given as a super-policy (described later). For an object encapsulating private information, the associated policy is given by a set of interfaces, using an augmented syntax where each interface has a *co-interface*,

describing the *who*-part and where the set of associated methods define the *how*-part. The co-interface gives restrictions on the *callee* object, only objects supporting the co-interface may call methods in the interface. Thus for a call $o.m(\ldots)$, the current object (this) must support the co-interface of a (super)interface of o that has m among its methods. For a function application $f(\ldots)$ resulting in type T with policy P, the resulting value must comply with P.

We allow *read* as a purpose, meaning all methods with read access. Similarly, purpose *increment* denotes all methods that do increment operations on the data, and purpose *read&incr* denotes all methods with read and increment access, see Figure [1]. The default policy triple $(subject; any; read)$ is implicitly included in every policy, where *subject* is the identity of the person who the private information is about. This gives the subject read access to information about himself. In order to make use of subject policy triples at static time, the static checking will try to detect if subject is the same as this or caller, considering also explicit testing. To enforce these policies, we extend Creol's type system. This way the policy accompanies the data, and support in complying with the obligations of *privacy by design* [Article 25, 1(a)] [2] and *privacy by default* [Article 25, 1(b)] [2].

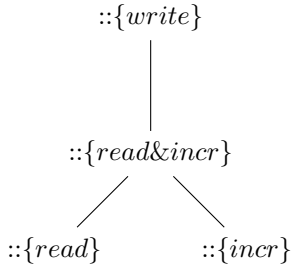


Figure 1: Access hierarchy

The static checking is done by a static typing system based on the kind of privacy policies outlined here. The rules involve the legal formation of expressions and function application as well as the imperative object-level where requirements to remote call are central. As mentioned interfaces use co-interfaces to specify roles, given by a **with** clause, and access is restricted by clauses such as :: *read* and :: *incr*, relative to a purpose.

Similarly, the policy for data types is specified by sets of policy triples of the form $(role; purpose; access)$. The static analysis includes static information about the *subject* of any private information. This approach is useful in avoiding inadvertent implementation mistakes and facilitate showing compliance with default privacy policies. We include a small example to illustrate the methodology, and its pictorial representation is given in Figure [2].

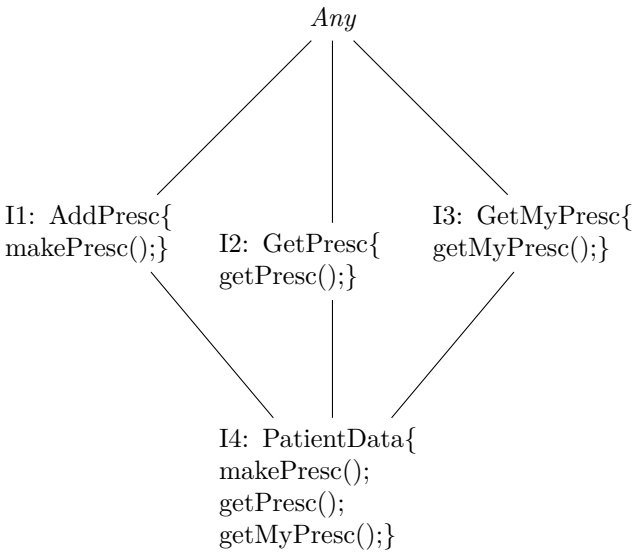


Figure 2: Healthcare Example

A Health Care Example

Here *Doctor*, *Nurse*, *Patient*, *PatientData* are interfaces. A *PatientData* object contains data for a number of patients, and can be accessed by doctors and nurses, based on different policies. Patients interact with Doctors and Nurses and may also access their own patient data. Patient data of type *Presc* is labeled with polices:$\{(doctor; treatment; read\&incr), (nurse; treatment; read)\}$, and implicitly $(subject; any; read)$. This policy allows *(i)* a patient to access his own data, *(ii)* gives read and increment access to the doctor for treatment purpose, and *(iii)* gives read-only access to the nurse for treatment purpose. *Treatment* is declared by the keyword **purpose** and interfaces can be annotated by such purposes, implying that all methods of the interface have that purpose. This allows an open ended specification of purposes, which is practically useful, allowing the purpose of policies made early depend on methods and interfaces declared later.

```
purpose treatment
interface AddPresc with Doctor{Void makePresc(Patient p, String presc)} in treatment
interface GetPresc with Nurse {String getPresc(Patient p)} in treatment
interface GetMyPresc with Patient {String getMyPresc()}
interface PatientData extends AddPresc, GetPresc, GetMyPresc {}
```

```
class PATIENTDATA() implements PatientData {
  type Presc == Map[Patient, List[String]]
     ::{(doctor; treatment; read&incr), (nurse; treatment; read)}
// here patient is the subject of the prescription data
  Presc presc := emptymap;
    with Doctor Void makePresc(Patient p, String newpresc) {presc[p] :+ newpresc;}
    with Nurse String getPresc(Patient p) {return last(presc[p]);}
    with Patient String getMyPresc() {return last(presc[caller]);} ... }
```

The co-interface of a method is given after the **with** keyword. It defines the minimal interface of a caller, which may be referred to by the *caller* parameter inside the method body. This allows us to talk about privileges of a caller. The use of co-interface ensures *callee* objects only with correct roles can invoke methods that access sensitive data. For example, if a nurse invokes makePresc() or getMyPresc(), the call would not pass the static test. Further details are omitted, including the privacy control of data types, such as maps ($Map[key, data]$). We use + for sequence append, and the statement $x :+ v$ abbreviates $x := x + v$ for x of a type with a + operation. Note that the *::incr* and *::read* restrictions are statically checked by inspecting changes on fields in the corresponding method bodies, allowing :+ in the former case.

References

[1] G. Danezis, J. Domingo-Ferrer, M. Hansen, J.-H. Hoepman, D. L. Metayer, R. Tirtea, and S. Schiffner, "Privacy and data protection by design-from policy to engineering," *arXiv preprint arXiv:1501.03726*, 2015.

[2] "European parliament and the council of the european union." Accessed: 2018-09-01.

[3] E. B. Johnsen, O. Owe, and I. C. Yu, "Creol: A type-safe object-oriented model for distributed concurrent systems," *Theoretical Computer Science*, vol. 365, no. 1-2, pp. 23–66, 2006.

[4] T. Ramezanifarkhani, O. Owe, and S. Tokas, "A secrecy-preserving language for distributed and object-oriented systems," *J. Log. Algebr. Meth. Program.*, vol. 99, pp. 1–25, 2018.

Graph Algebras and Software Engineering

Uwe Wolter, Department of Informatics, University of Bergen, Norway

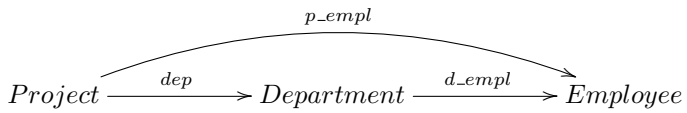
The latest trend in software engineering regards models as first-class entities of the software development process. This trend has led to a branch of software engineering, often called Model-Driven Engineering (MDE), which promotes modelling as the main activity of software development and pursues the shift of paradigm from code-centric to model-centric.

The different kinds of models, exploited in Software Engineering and, especially, in MDE, appear often as graph-based structures. Generalized Sketches [Mak97, DK97] are a universal and flexible mathematical specification formalism for those graph-based structures. Starting from generalized sketches, we developed in Bergen, during the last decade, the so-called Diagram Predicate Framework (DPF) to describe different kinds of graph-based structures and their semantics in a uniform way, thus providing a formal ground to relate, extend and integrate different diagrammatic modelling techniques and to define and implement meta-modelling, version control and model transformations, for example [DW08, RRLW10, RRLW12, RdLG$^+$14].

Operations on graph-based structures are omnipresent in Software Engineering and, especially, in MDE. However, a proper and adequate formalization of those operations has been missing until lately. As a first essential step towards such a formalization, we introduced in [WDK18] the concept of **graph algebra** that generalizes the traditional concept of algebra.

Graph algebras are not "algebras of graphs", as they have been described and studied in [CG99, BCG$^+$10], for example. "Algebras of graphs" are special traditional algebras where the carrier is a set of (finite) graphs. In contrast, the carrier of a graph algebra is one single **graph**[1] that will be often infinite in applications. **Graph operations**, i.e., operations in a graph algebra, describe how certain finite parts of the carrier graph can be "computed" based on other finite parts of the carrier graph. In traditional algebra the arity of an operation is described by a finite set of "input positions", while the input and output arity, respectively, of a graph operation is described by a finite graph.

To illustrate the concept of graph algebra and its potential role in Software Engineering, we consider a graph D representing a class diagram with three classes *Project*, *Department*, *Employee* and three references *dep*, *d_empl*, *p_empl*:



The reference *dep* provides for each project the departments controlling it. *d_empl* gives us for each department all the employees working for this department while *p_empl* informs us about all the employees involved in a certain project.

To describe the semantics of 'class diagrams with references', we can utilize a graph algebra $\mathcal{M}$ where the nodes in the corresponding infinite carrier graph M are finite sets A (of objects) and the edges $A \xrightarrow{f} B$ in M represent *multimaps* (set-valued functions) $f : A \to \wp(B)$ between finite sets. The semantics of the class diagram D, at a certain point in time, will be then a certain interpretation of D in M, i.e., a **graph homomorphism**[2] $state : \mathsf{D} \to \mathsf{M}$. This "state" may change over time.

One thing we can do in class diagrams is to navigate through paths of references. This kind of navigation is semantically based on the composition of multimaps[3]. Therefore, our graph algebra $\mathcal{M}$ should

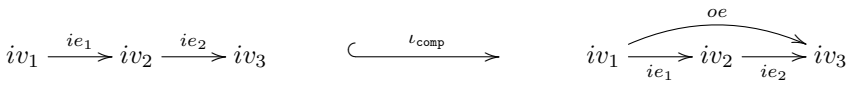
[1] A graph $\mathsf{G} = (G_V, G_E, sc^{\mathsf{G}}, tg^{\mathsf{G}})$ consists of a set G_V of nodes, a set G_E of edges, and two maps $sc^{\mathsf{G}}, tg^{\mathsf{G}} : G_E \to G_V$.

[2] A homomorphism $\varphi = (\varphi_V, \varphi_E)$ between two graphs $\mathsf{G} = (G_V, G_E, sc^{\mathsf{G}}, tg^{\mathsf{G}})$ and $\mathsf{H} = (H_V, H_E, sc^{\mathsf{H}}, tg^{\mathsf{H}})$ consists of two maps $\varphi_V : G_V \to H_V$ and $\varphi_E : G_E \to H_E$ such that $sc^{\mathsf{G}}; \varphi_V = \varphi_E; sc^{\mathsf{H}}$ and $tg^{\mathsf{G}}; \varphi_V = \varphi_E; tg^{\mathsf{H}}$.

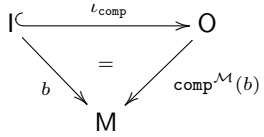
[3] The composition $f; g : A \to \wp(C)$ of two multimaps $f : A \to \wp(B)$ and $g : B \to \wp(C)$ is defined by $f; g(a) := \bigcup\{g(b) \mid b \in f(a)\}$ for all $a \in A$.

comprise, besides other operations, a composition operation. An input of the composition operation should be a sequence $A \xrightarrow{f} B \xrightarrow{g} C$ of two connected edges in M while the output should be an edge from A to C.

This means that we declare in the signature for our graph algebra $\mathcal{M}$ an operation symbol $\mathtt{comp}$ together with an input arity graph I and an output arity graph O



where we include, for simplicity reasons, the whole graph I into the graph O. The corresponding graph operation $\mathtt{comp}^{\mathcal{M}}$ in $\mathcal{M}$ computes than for each "actual input" (binding), i.e., for each graph homomorphism $b : \mathsf{I} \to \mathsf{M}$, a unique "actual output", i.e., a graph homomorphism $\mathtt{comp}^{\mathcal{M}}(b) : \mathsf{O} \to \mathsf{M}$. Operations shouldn't have side effects thus we require that the following triangle of graph homomorphisms commutes

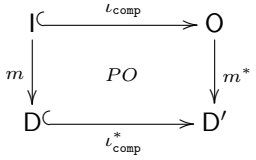


The commutativity requirement entails that the composition operation needs to compute only the "edge value" $\mathtt{comp}^{\mathcal{M}}(b)(oe)$ and that this "edge value" should be an edge from $\mathtt{comp}^{\mathcal{M}}(b)(iv_1) = b(iv_1)$ to $\mathtt{comp}^{\mathcal{M}}(b)(iv_3) = b(iv_3)$. We can define now the graph operation $\mathtt{comp}^{\mathcal{M}}$ by means of the composition of multimaps:

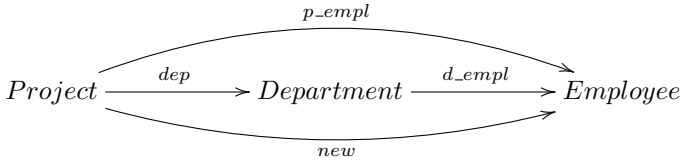
$$\mathtt{comp}^{\mathcal{M}}(b)(oe) := b(ie_1); b(ie_2) \quad \text{for all } b : \mathsf{I} \to \mathsf{M}.$$

However, what about to represent composition syntactically on the level of class diagrams. In our example, we could introduce a new edge from *Project* to *Employee* to represent the composition of *dep* and *d_empl*, for example.

Formally this can be described by (1) considering the declaration of the arity of $\mathtt{comp}$, i.e., the inclusion graph homomorphism $\iota_{\mathtt{comp}} : \mathsf{I} \hookrightarrow \mathsf{O}$, as a graph transformation rule, (2) choosing a match for the left-hand side I of the rule in the present graph D, i.e., a graph homomorphism $m : \mathsf{I} \to \mathsf{D}$, and (3) constructing an extension D' of D by constructing the pushout of the span $\mathsf{D} \xleftarrow{m} \mathsf{I} \xrightarrow{\iota_{\mathtt{comp}}} \mathsf{O}$ of graph homomorphisms



In our example we chose the (only) match $m : \mathsf{I} \to \mathsf{D}$ uniquely represented by the assignment $(ie_1 \mapsto dep, ie_2 \mapsto d\_empl)$. The corresponding rule application introduces then a new fresh edge from *Project* to *Employee* representing the composition of *dep* and *d_empl*. The derived reference *new* informs us about all the employees in any of the departments controlling a certain project.



This derived reference can be used now, for example, to formalize the constraint "An employee involved in a project must work in one of the controlling departments" (see [RRLW12]).

It may be not obvious, but the pushout construction is just a straightforward generalization of the construction of terms in the traditional algebraic setting: Given an n-ary operation symbol ω and a

binding for the corresponding n "input positions", i.e., an n-tuple $(t_1, \ldots, t_n)$ of terms, we generate a new term $\omega\langle t_1, \ldots, t_n\rangle$. The crucial observation is that the syntactic expression $\omega\langle t_1, \ldots, t_n\rangle$ serves two purposes. First, it encodes the information what "rule" and what "match" has been used to generate the new data item. Second, by doing so, it creates a "fresh name" identifying the new data item uniquely.

In the setting of graph algebras an operation application will, in general not compute a single data item but one data item for each "output item" in $\mathsf{O} \setminus \mathsf{I}$. So, in this new setting we have to use syntactic expressions encoding as well the "rule" and the "match" as the "output item" in $\mathsf{O} \setminus \mathsf{I}$ to identify generated new data items uniquely (compare [WDK18]). In our example we could use an expression like $\langle oe, \mathtt{comp}, \langle ie_1 \mapsto dep, ie_2 \mapsto d\_empl\rangle\rangle$ or its shorthand $\langle oe, \mathtt{comp}, \langle dep, d\_empl\rangle\rangle$, for example, to denote the fresh edge *new* uniquely.

In full analogy to terms, any interpretation $state : \mathsf{D} \to \mathsf{M}$ of the original class diagram D extends uniquely to an interpretation $state : \mathsf{D}' \to \mathsf{M}$ of the extended class diagram where

$$state(new) = state(\langle oe, \mathtt{comp}, \langle dep, d\_empl\rangle\rangle) = \mathtt{comp}^{\mathcal{M}}(state(dep), state(d\_empl))(oe).$$

The derived reference $new = \langle oe, \mathtt{comp}, \langle dep, d\_empl\rangle\rangle$ enables as now to check if the constraint "An employee involved in a project must work in one of the controlling departments" is satisfied by the present state $state : \mathsf{D} \to \mathsf{M}$ of the original (!) class diagram D:

$$state(p\_empl)(P) \subseteq state(new)(P) \quad \text{for all } P \in state(Project).$$

References

[BCG+10] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and U. Montanari. On GS-Monoidal Theories for Graphs with Nesting. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model-Driven Engineering*, volume 5765 of *Lecture Notes in Computer Science*, pages 59–86. Springer, 2010.

[CG99] A. Corradini and F. Gadducci. An Algebraic Presentation of Term Graphs, via GS-Monoidal Categories. *Applied Categorical Structures*, 7:299–331, 1999.

[DK97] Z. Diskin and B. Kadish. A graphical yet formalized framework for specifying view systems. In *Advances in Databases and Information Systems*, pages 123–132, 1997. ACM SIGMOD Digital Anthology: vol.2(5), ADBIS'97.

[DW08] Zinovy Diskin and Uwe Wolter. A Diagrammatic Logic for Object-Oriented Visual Modeling. *ENTCS*, 203/6:19–41, 2008.

[Mak97] M. Makkai. Generalized sketches as a framework for completeness theorems. *Journal of Pure and Applied Algebra*, 115:49–79, 179–212, 214–274, 1997.

[RdLG+14] Alessandro Rossini, Juan de Lara, Esther Guerra, Adrian Rutle, and Uwe Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, pages 1–38, 2014.

[RRLW10] Alessandro Rossini, Adrian Rutle, Yngve Lamo, and Uwe Wolter. A formalisation of the copy-modify-merge approach to version control in MDE. *Journal of Logic and Algebraic Programming*, 79(7):636–658, 2010.

[RRLW12] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in MDE. *Journal of Logic and Algebraic Programming*, 81/4:422–457, 2012.

[WDK18] Uwe Wolter, Zinovy Diskin, and Harald König. Graph Operations and Free Graph Algebras. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*, pages 313–331. Springer, LNCS 10800, 2018.

Adding Constraints to CRDTs: From Eventual Consistency to Observable Atomic Consistency*

Xin Zhao and Philipp Haller

KTH Royal Institute of Technology
Stockholm, Sweden
{xizhao,phaller}@kth.se

Abstract

Conflict-free replicated data types (CRDTs) are widely used in eventually consistent systems to reduce concurrency control. However, it is not always suitable for applications which require stronger consistency. Thus, programmers need to constrain the consistency levels on their own, increasing the possibilities of programming errors. In this paper, we introduce a method which enhances the consistency level of CRDTs by adding constraint operations. Our method provides observable atomic consistency as an enhancement for otherwise eventually-consistent CRDTs. We also provide a high-level programming interface to improve the efficiency and accuracy of distributed programming.

1 Introduction

Conflict-free replicated data types (CRDTs) [4, 3] are widely used in industrial distributed systems. The property of the data types is mathematically proven to be able to merge or resolve concurrent updates. Thus, CRDTs are ideal for achieving high availability for replicated shared data and also guarantee strong eventual consistency. However, the primary challenge of programming with CRDTs is that they only ensure eventual consistency and are restricted in the defined operations. For example, a read operation may return outdated data before the replicas finally converge, which might cause program errors and thus needs to be taken care of by the application developers themselves.

In this paper, we address this challenge by extending CvRDTs with additional constraint operations, called totally-ordered operations. The idea is to provide a method to force data to merge when a specific operation is invoked. If all replicas remain consistent up to this point, we say the system guarantees observable atomic consistency. We first give a motivational example in Section 2, then introduce observable atomic consistency in Section 3, and explain the user interface in Section 4; Section 5 summarizes related work, and Section 6 concludes.

2 Motivation

One of the typical examples of CRDTs is the grow-only counter (also called GCounter). It is widely used in real-time analytics and distributed gaming. However, it only supports increment and mergable operations which limits the application scenarios; e.g., it lacks a "reset" operation. "Reset" is necessary for setting the counter back to its initial default state. In current popular implementations, such as Riak DT, this is not supported, though.

*This extended abstract is based on a paper [5] accepted for publication at the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2018).

Thus, we need an implementation of a resettable counter to set the counter values in all the replicas to the initial state. The "reset" operation needs to be defined as a particular operation for CRDTs, so that the state can be forced to be reset.

In particular, when the system invokes the "reset" operation, it needs to make sure that all the replicas in the system reach the same state before any other operations take effect. Thus, it is necessary to define a new consistency model for the system to handle such an additional operation. We introduce such a consistency model in the following.

3 Observable Atomic Consistency

Now we introduce the definition of observable atomic consistency which is used to restrict the consistency levels for additional operations. The content of this section is mainly taken from our forthcoming paper [5].

Definition 3.1 (CvT order)**.** Given a set of operations $U = C \cup T$ where $C \cap T = \varnothing$, a CvT order is a partial order $O = (U, \prec)$ with the following restrictions:

- $\forall u, v \in T$ such that $u \neq v$. $u \prec v \vee v \prec u$
- $\forall p \in C$, $u \in T$. $p \prec u \vee u \prec p$
- $\forall l, m, n \in U$ such that $l \prec m, m \prec n$. $l \prec n$

Definition 3.2 (Cv-set)**.** Given a set of operations $U = C \cup T$ where $C \cap T = \varnothing$, a Cv-set C_i is a set of C operations with the restriction that:

- $\forall p, q \in C_i \Rightarrow p \not\prec q \wedge q \not\prec p$
- $\forall p \in C \setminus C_i$. $\exists q \in C_i$ such that $p \prec q \vee q \prec p$

In the definitions above, the set of operations C stands for the set of original CRDT operations and the set T stands for the set of additional totally-ordered operations.

Definition 3.3 (Local atomic consistency (LAC))**.** A replicated system provides local atomic consistency (LAC) if each site i applies all operations according to a linear extension of the CvT order.

Definition 3.4 (Observable atomic consistency)**.** A replicated system provides observable atomic consistency (OAC) if it provides local atomic consistency and for all $p \in C$, $u \in T$. (a) $p \prec_{PO} u$ in program order (of some client) implies that $p \prec u$ in the CvT order, and (b) $u \prec_{PO} p$ in program order (of some client) implies that $u \prec p$ in the CvT order.

A complete explanation of the concepts above and a proof of state convergence can be found in our companion technical report [6].

4 Implementation and User Interface

Observable atomic consistency provides a stronger consistency guarantee for additional totally-ordered operations than eventual consistency for CRDTs while achieving higher availability for CRDT operations than atomic consistency. Due to the space limitations, we include the consistency protocol and the implementation based on the Akka [2] framework in our paper [5].

Here we show a small example of the user interface for using the additional constraint operations.

```
class CounterClient extends Protocol[GCounter] {
  val CounterClientBehavior: Receive = {
    case Incr  => self forward CvOp("incr")
    case Reset => self forward TOp("Reset")
    ...
  }
}
```

For the user interface, the additional constraint totally-ordered operations such as "Reset" are sent via the `TOp()` message handler and the system guarantees OAC as defined in Section 3, which hides the implementation details from the application developer.

5 Related Work

A closely related consistency model which also enhances eventual consistency is RedBlue consistency [1]. Red and blue operations are two kinds of operations in their system. Red operations are totally ordered while the blue ones can commute globally. In this sense, RedBlue consistency maintains a local view of the system. In OAC, the CvRDT updates can only commute inside a specific scope, namely, within one Cv-set. This restricts the flexibility of CvRDT updates, but at the same time provides a globally consistent view of the state.

6 Conclusion and Future Work

We devised a novel "observable atomic consistency (OAC)" model for enhancing the consistency level of additional operations for CRDTs. OAC provides a consistent view of the system and guarantees correct system behavior as long as developers correctly classify their operations. In our companion technical report, we also show in the experimental evaluation that the system providing OAC significantly reduces coordination compared with atomic consistency while maintaining the availability of convergent operations. In future work, we would like to provide operational semantics of the protocol, enabling proofs for distributed programs using OAC. Moreover, we would like to investigate type systems enabling safe transitions between different consistency levels.

References

[1] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278. USENIX Association, 2012.

[2] Lightbend, Inc. Akka. http://akka.io/, 2009. Accessed: 2016-03-20.

[3] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical Report RR-7506; inria-00555588, HAL CCSD, January 2011.

[4] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

[5] Xin Zhao and Philipp Haller. Observable atomic consistency for CvRDTs. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2018*. ACM, 2018. to appear.

[6] Xin Zhao and Philipp Haller. Observable atomic consistency for CvRDTs (extended version). *CoRR*, abs/1802.09462, 2018.