31st Nordic Workshop on Programming Theory

NWPT 2019

Tallinn, Estonia, 13–15 November 2019

Abstracts

Department of Software Science, Tallinn University of Technology

Tallinn ◦ 2019

31st Nordic Workshop on Programming Theory
NWPT 2019
Tallinn, Estonia, 13–15 November 2019
Abstracts

Edited by Tarmo Uustalu and Jüri Vain

Department of Software Science, Tallinn University of Technology
Akadeemia tee 15A, 12618 Tallinn, Estonia

ISBN 978-9949-83-520-1 (pdf)

Preface

This volume contains the abstracts of the talks to be presented at the 31st Nordic Workshop on Programming Theory, NWPT 2019, to take place in Tallinn, Estonia, 13–15 November 2019.

The NWPT workshops are a forum bringing together programming theorists from the Nordic and Baltic countries (but also elsewhere). The previous workshops were held in Uppsala (1989, 1999 and 2004), Aalborg (1990 and 2016), Göteborg (1991 and 1995), Bergen (1992, 2000 and 2012), Åbo (Turku) (1993, 1998, 2003, 2010 and 2017), Aarhus (1994), Oslo (1996, 2007 and 2018), Tallinn (1997, 2002, 2008 and 2013), Lyngby near Copenhagen (2001 and 2009), Copenhagen (2005), Reykjavík (2006 and 2015), Västerås (2011) and Halmstad (2014).

The scope of the meetings covers traditional as well as emerging disciplines within programming theory: semantics of programming languages, programming language design and programming methodology, programming logics, formal specification of programs, program verification, program construction, program transformation and refinement, real-time and hybrid systems, models of concurrent, distributed and mobile computing, language-based security. In particular, they are targeted at early-career researchers as a friendly forum where one can present work in progress but which at the same time produces a high-level post-proceedings compiled of the selected best contributions in the form of a special journal issue.

The programme of NWPT 2019 includes three invited talks by distinguished researchers—Paweł Sobociński (Tallinn University of Technology), Ando Saabas (Bolt) and Jan von Plato (University of Helsinki). The contributed part of the programme consists of 23 talks by authors from Belgium, Estonia, Finland, Germany, Iceland, Norway, Poland, Russia, Sweden and the United Kingdom.

Tarmo Uustalu and Jüri Vain

Tallinn, 11 November 2019

Organization

Programme Committee

Antonis Achilleos (Reykjavik University)
Johannes Borgström (Uppsala University)
Martin Elsman (University of Copenhagen)
Daniel S. Fava (University of Oslo)
John Gallagher (Roskilde University)
Michael R. Hansen (Technical University of Denmark)
Magne Haveraaen (University of Bergen)
Keijo Heljanko (University of Helsinki)
Thomas T. Hildebrandt (University of Copenhagen)
Einar Broch Johnsen (University of Oslo)
Jaakko Järvi (University of Bergen)
Yngve Lamo (Western Norway University of Applied Sciences)
Alberto Lluch Lafuente (Technical University of Denmark)
Fabrizio Montesi (University of Southern Denmark)
Wojciech Mostowski (Halmstad University)
Olaf Owe (University of Oslo)
Philipp Rümmer (Uppsala University)
Gerardo Schneider (University of Gothenburg)
Cristina Seceleanu (Mälardalen University)
Jiří Srba (Aalborg University)
Tarmo Uustalu (Reykjavik University / Tallinn University of Technology)
Jüri Vain (Tallinn University of Technology)
Antti Valmari (University of Jyväskylä)
Marina Waldén (Åbo Akademi University)

Additional Reviewers

Duncan Paul Attard
Jaan Priisalu
Morten Rhiger
Tiberiu Seceleanu
Leonidas Tsiopoulos
Niccolò Veltri

Organizing Committee

Anu Baum (Tallinn University of Technology)
Juhan Ernits (Tallinn University of Technology)
Marko Kääramees (Tallinn University of Technology)
Monika Perkmann (Tallinn University of Technology)
Tarmo Uustalu (Reykjavik University / Tallinn University of Technology)
Jüri Vain (Tallinn University of Technology)

Host

Department of Software Science, Tallinn University of Technology

Sponsor

Research measure of the Estonian IT Academy programme

Table of Contents

Invited talks

Contributed talks

Interpreting and Validating Machine Learning Models

Ando Saabas

Bolt, Tallinn, Estonia
ando@set.ee

Machine learning (ML) models are being increasingly used in practical applications, including in fields such as healthcare, manufacturing and transportation. These models, while powerful, can sometimes fail in very unintuitive ways. As the size and complexity of the models increases, it has become more and more important to understand the reasoning of the models, i.e., to validate that the interplay between specification (read: training data) and program (read: the model) is what was intended.

In this talk, I will give an overview of the state of the art in interpreting ML models. In particular, I will explain how tree-based models such as random forests or gradient boosted trees can be instrumented to explain their predictions in terms of contributions from each feature in the input feature vector.

A Compositional Approach to Signal Flow Graphs

Paweł Sobociński

Dept. of Software Science, Tallinn University of Technology, Estonia
pawel@cs.ioc.ee

Signal flow graphs are a classical state-machine model of computation first proposed by Claude Shannon, and are well-known in control theory and engineering. They have a continuous interpretation, where they compute solutions of systems of homogeneous higher-order differential equations, and a discrete interpretation, where they compute solutions of recurrence relations.

I will give a compositional presentation of the theory of signal flow graphs using standard techniques of programming language semantics. String diagrams provide a rigorous graphical syntax, and a denotational semantics is given as a monoidal functor to an appropriate category of linear relations. Operational semantics can be given using a structural presentation.

Denotational equality will be characterised: i) axiomatically, in terms of an equational theory that shows how the basic syntactic components "interact", ii) and in operational terms as contextual equivalence, via a full abstraction result that relies on recent work that extends the setting from linear to affine relations.

Two Centuries of Formal Computation

Jan von Plato

Dept. of Philosophy, University of Helsinki, Finland
jan.vonplato@helsinki.fi

Theories of formal computation preceded actual programmable computers by about one hundred years. The first intimations of such computation go back even further, to one Johann Schultz, professor of mathematics and royal court-preacher in Kant's Königsberg, and to Leibniz. Google Books and other online sources have made it possible to illustrate through original sources the long way from Leibniz' formal proof of $2 + 2 = 4$ to the 1930s that represented formal computation as a species of formal deduction.

Axiomatizing Equivalences over Regular Monitors*

Luca Aceto[1,2], Antonis Achilleos[1], Elli Anastasiadi[1], Anna Ingólfsdóttir[1]

[1] Dept. of Computer Science, Reykjavik University, Iceland
[2] Gran Sasso Science Institute, L'Aquila, Italy
luca@ru.is, luca.aceto@gssi.it, antonios@ru.is, elli19@ru.is, annai@ru.is

Abstract

We study whether recursion-free regular monitors have finite equational axiomatizations with respect to two notions of equivalence, namely verdict and ω-verdict equivalence.

1 Introduction

Equational axiomatizations provide a purely syntactic description of the chosen notion of equivalence over processes and characterize the essence of a process semantics by means of a few revealing axioms. We consider a fragment of the regular monitors described and studied by Aceto et al. in, for instance, [1, 4]. Monitors are a key tool in the field of runtime verification [3], where they are used to check for system properties by analyzing execution traces generated by processes.

A monitor is an agent that observes the events occurring in a system as it progresses through time. Two monitors are verdict equivalent when they characterize exactly the same traces as successful and failure ones. Similarly they are ω-verdict equivalent when they characterize the same infinite traces as successful and failure ones. Our goal in this work is to study the equational theory of those relations. We give a ground complete axiomatization for both of those equivalences. We also study open equations, provide an (infinite) complete axiomatization for verdict equivalence and we argue that no finite one exists, even when the set of actions monitors can analyze is finite. Such negative results are common in the field of process algebra [2] although they usually occur for more expressive languages.

We present a suitable notion of *normal form* and use it to reduce the verdict equivalence problem for closed monitors to equality between normal forms. We also study the complexity of checking equivalence between two closed monitors and find it to be almost linear (off by a constant factor) in the size of the syntax tree of the monitors. This result is in contrast with the $coNP$-completeness for equality testing between star-free regular expressions [6].

Apart from their intrinsic theoretical interest, axiomatizations such as the ones we present can form the basis for tools for proving equivalences between monitors using theorem-proving techniques and identify valid laws of "monitor programming" in the sense of [5]. A complete axiomatization captures all the valid laws of programming in a model-independent way. Such laws can, for instance, be used as a set of rewrite rules to bring a monitor into an equivalent but better (for instance, more succinct or canonical) form. Since monitors are often synthesized automatically from specifications of monitorable properties, non-optimal representations are very likely to arise as a result of monitor-synthesis algorithms. The availability of a complete axiomatization of monitor equivalence indicates that, at least for monitors written in the languages being axiomatized modulo the chosen notion of equivalence, one can always synthesize "optimal" monitors.

*The work reported in this paper is supported by the projects Open Problems in the Equational Logic of Processes (OPEL) (grant 196050-051) and 'TheoFoMon: Theoretical Foundations for Monitorability' (grant 163406-051) of the Icelandic Research Fund. Acetos work was also partially supported by the Italian MIUR PRIN 2017FTXR7S project IT MATTERS 'Methods and Tools for Trustworthy Smart Systems'.

2 Background

Syntax of monitors Let Act be a set of visible actions, ranged over by a, b. Following Milner [7], we use $\tau \notin Act$ to denote an unobservable action. We will denote the set of infinite sequences over Act as Act^ω. As usual Act^* stands for the set of finite sequences over Act. Let Var be a countably infinite set of variables, ranged over by x, y, z.

The collection Mon_F of regular, recursion-free monitors is the set of terms generated by the following grammar:

$$m, n ::= \; v \;\mid\; a.m \;\mid\; m + n \;\mid\; x \qquad\qquad v ::= end \mid yes \mid no,$$

where $a \in Act$ and $x \in Var$. The terms *end*, *yes* and *no* are called *verdicts*. Closed monitors are those that do not contain any occurrences of variables. For each $\alpha \in Act \cup \{\tau\}$, we define the transition relation $\xrightarrow{\alpha} \subseteq Mon_F \times Mon_F$ as the least one that satisfies the following rules:

$$\frac{}{a.m \xrightarrow{a} m} \qquad \frac{m \xrightarrow{\alpha} m'}{m + n \xrightarrow{\alpha} m'} \qquad \frac{n \xrightarrow{\alpha} n'}{m + n \xrightarrow{\alpha} n'} \qquad \frac{}{v \xrightarrow{\alpha} v} \, .$$

For $s = a_1 a_2 \ldots a_n \in Act^*$ and $n \geq 0$, we use $m \overset{s}{\Rightarrow} m'$ to mean that:

1. $m(\xrightarrow{\tau})^* m'$ if $s = \varepsilon$, where ε stands for the empty string,
2. $m \overset{\varepsilon}{\Rightarrow} m_1 \xrightarrow{a} m_2 \overset{\varepsilon}{\Rightarrow} m'$ for some m_1, m_2 if $s = a \in Act$ and
3. $m \overset{a}{\Rightarrow} m_1 \overset{s'}{\Rightarrow} m'$ for some m_1 if $s = a.s'$.

If $m \overset{s}{\Rightarrow} m'$ for some m', we call s a trace of m.

Verdict and ω-Verdict Equivalence Let m be a (closed) monitor. We define:

$$L_a(m) = \{s \in Act^* \mid m \overset{s}{\Rightarrow} yes\} \text{ and } L_r(m) = \{s \in Act^* \mid m \overset{s}{\Rightarrow} no\}.$$

Note that we allow for monitors that may both accept and reject some trace. This is necessary to maintain our monitors closed under +. Of course, in practice, one is interested in monitors that are consistent in their verdicts.

Definition 1. *Let m and n be closed monitors. We say that m and n are **verdict equivalent**, written $m \simeq n$, iff $L_a(m) = L_a(n)$ and $L_r(m) = L_r(n)$. We say that m and n are ω-**verdict equivalent**, written $m \simeq_\omega n$, iff $L_a(m) \cdot Act^\omega = L_a(n) \cdot Act^\omega$ and $L_r(m) \cdot Act^\omega = L_r(n) \cdot Act^\omega$. These equivalences are extended to open monitors in the standard way.*

Lemma 1. *The following hold: (1) $\simeq$ and $\simeq_\omega$ are both congruences. (2) $\simeq \;\subseteq\; \simeq_\omega$ and the inclusion is strict when Act is finite. (3) If Act is infinite then $\simeq \;=\; \simeq_\omega$.*

- $A1$: $x + y = y + x$
- $A2$: $x + (y + z) = (x + y) + z$
- $A3$: $x + x = x$
- $A4$: $x + end = x$
- $E1_a$: $a.end = end$ $(a \in Act)$
- Y_a: $yes = yes + a.yes$ $(a \in Act)$
- N_a: $no = no + a.no$ $(a \in Act)$
- $D1_a$: $a.(x + y) = a.x + a.y$ $(a \in Act)$

3 Results on axiomatizations and complexity

Our axiom system for verdict equivalence over closed monitors is $\mathcal{E}_{veq}$, whose axioms are:

Theorem 1. *$\mathcal{E}_{veq}$ is complete over closed monitors modulo $\simeq$. That is if m, n are closed monitors in Mon_F and $m \simeq n$ then $\mathcal{E}_{veq} \vdash m = n$. Moreover, $\mathcal{E}_{veq}$ is complete over closed terms modulo $\simeq_\omega$ when Act is infinite.*

In order to capture ω-verdict equivalence when Act is finite, we extend the axiom set by: $\mathcal{E}_{\omega-veq} = \mathcal{E}_{veq} \cup \{yes = \sum_{a \in Act} a.yes\} \cup \{no = \sum_{a \in Act} a.no\}$. We then prove:

Theorem 2. *$\mathcal{E}_{\omega-veq}$ is complete over closed terms modulo $\simeq_\omega$. That is if m, n are closed monitors in Mon_F and $m \simeq_\omega n$ then $\mathcal{E}_{\omega-veq} \vdash m = n$,when Act is finite.*

Naturally we continue towards the relevant results for open terms. Initially we only expand the axioms as: $\mathcal{E}'_{veq} = \mathcal{E}_{veq} \cup \{yes + no + x = yes + no\}$ and then prove:

Theorem 3. *$\mathcal{E}'_{veq}$ is complete for open terms modulo $\simeq$ when Act is infinite. That is, if m, n are open monitors in Mon_F and $m \simeq n$ then $\mathcal{E}'_{veq} \vdash m = n$.*

The final part of this work is dedicated to determining an axiomatization with respect to $\simeq$ and $\simeq_\omega$ for open equations when Act is finite and it includes an analysis on why the axiomatization cannot be finite.

Finally regarding the complexity of determining whether two monitors are equivalent, our completeness proof suggests that discovering a normal form for a monitor implies a somewhat pre-defined application of our axioms. In the case of closed terms the "normalization" procedure with respect to verdict equivalence can take place recursively and in a way mimicking the inductive proof of the existence of a normal form. After this is done then the equality testing of two monitors is trivially testing equality for two rooted, ordered and labelled trees. The final complexity of the algorithm is $O(n \cdot k \cdot log(k))$ where k is the size of Act and n is the sum of the sizes of the syntactic trees of the tested monitors. Future work includes the complexity analysis for verdict and ω-verdict equivalence testing between open monitors and the study of the equational theory of regular monitors with recursion.

References

[1] Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: From branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL):52:1–52:29, 2019.

[2] Jos C.M. Baeten, Twan Basten, and Michel A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2009.

[3] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification: Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[4] Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the Hennessy–Milner Logic with recursion. *Form. Methods Syst. Des.*, 51(1):87–116, 2017.

[5] C. A. R. Hoare, Ian J. Hayes, Jifeng He, Carroll Morgan, A. W. Roscoe, Jeff W. Sanders, Ib Holm Sørensen, J. Michael Spivey, and Bernard Sufrin. Laws of programming. *Commun. ACM*, 30(8):672–686, 1987.

[6] Harry B. Hunt, Daniel J. Rosenkrantz, and Thomas G. Szymanski. On the equivalence, containment, and covering problems for the regular and context-free languages. *J. Comput. Syst. Sci.*, 12:222–268, 1976.

[7] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

Axiomatising Weighted Monadic Second-Order Logic on Finite Words

Antonis Achilleos and Mathias Ruggaard Pedersen

Dept. of Computer Science, Rekjavik University, Iceland
{antonios,mathiasrp}@ru.is

1 Introduction

Weighted automata constitute a popular and useful framework for specifying and modelling the behaviour of quantitative systems. In order to reason about such systems, a logical description language is often used. One such logic is monadic second-order logic (MSO), which is known to describe exactly the behaviour of unweighted automata by a theorem of Büchi, Elgot, and Trakhtenbrot [1, 4, 10]. A more recent result by Droste and Gastin [2] showed that a weighted extension of MSO captures the behaviour of weighted automata in a similar fashion. While this result has been extended in many different ways, a proper analysis of this logic in the form of axiomatisation, satisfiability, and model checking issues has not yet surfaced.

One of the difficulties of such an analysis is that the weighted MSO (wMSO) is interpreted over an arbitrary set of values, rather than a standard Boolean setting. This means that each formula is a function which takes a model and returns a value, rather than something which may or may not be satisfied by a given model. We therefore investigate how to extend familiar concepts such as completeness, validity, and satisfiability to this non-Boolean, real-valued setting.

We document here some of our on-going work on these issues, including presenting equational systems that give a complete axiomatisation of a fragment of weighted MSO as well as algorithms for some of the variants of satisfiability checking in this setting.

2 Syntax and Semantics of wMSO

Following [5], we define the syntax and semantics of wMSO as follows. Consider a finite set of first-order variables $\mathcal{V}_{\mathsf{FO}}$, a finite set of second-order variables $\mathcal{V}_{\mathsf{MSO}}$, a finite alphabet Σ, and an arbitrary set $\mathcal{R}$ of weights. Note that we assume no structure on $\mathcal{R}$, it does not even have to be a semiring. The syntax of wMSO is given by the following grammar, divided into two layers.

$$\varphi ::= \top \mid P_a(x) \mid x \leq y \mid x \in X \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \forall x\varphi \mid \forall X\varphi \qquad (\mathsf{MSO})$$

$$\Psi ::= r \mid \varphi \,?\, \Psi_1 : \Psi_2 \qquad (\mathsf{wMSO})$$

Here, $a \in \Sigma$, $r \in \mathcal{R}$, $x, y \in \mathcal{V}_{\mathsf{FO}}$, and $X \in \mathcal{V}_{\mathsf{MSO}}$. The first layer, MSO, is simply MSO on finite words. The second layer, wMSO, allows one to condition on MSO formulas, and choose different values of $\mathcal{R}$ depending on the truth of the MSO formulas.

The MSO formulas are interpreted in the standard way over words $w \in \Sigma^+$ together with a valuation of this word σ, which assigns to each first-order variable a position in the word and to each second-order variable a set of such positions. We denote by Σ^+_σ the set of pairs (w, σ) where $w \in \Sigma^+$ and σ is a valuation of w, and we denote by $[\![\varphi]\!]$ the set of all pairs $(w, \sigma) \in \Sigma^+_\sigma$ that satisfies φ.

$$\Gamma \vdash r \approx r \qquad \Gamma \vdash \Psi \approx \varphi \,?\, \Psi : \Psi \qquad \Gamma \vdash \neg\varphi \,?\, \Psi_1 : \Psi_2 \approx \varphi \,?\, \Psi_2 : \Psi_1$$
$$\Gamma \cup \{\varphi\} \vdash \Psi_1 \approx \Psi_2, \quad \text{if } \Gamma \vdash \Psi_1 \approx \Psi_2 \qquad \Gamma \vdash \varphi \,?\, \Psi_1 : \Psi_2 \approx \Psi_1, \quad \text{if } \Gamma \vdash \varphi \leftrightarrow \top$$
$$\Gamma \vdash \varphi \,?\, \Psi_1 : \Psi_2 \approx \Psi, \quad \text{if } \Gamma \cup \{\varphi\} \vdash \Psi_1 \approx \Psi \text{ and } \Gamma \cup \{\neg\varphi\} \vdash \Psi_2 \approx \Psi$$

Table 1: Axioms for wMSO.

The semantics of formulas Ψ of wMSO is given by a function $[\![\cdot]\!] : \Sigma_\sigma^+ \to \mathcal{R}$, defined by

$$[\![r]\!]\,(w,\sigma) = r \qquad [\![\varphi \,?\, \Psi_1 : \Psi_2]\!]\,(w,\sigma) = \begin{cases} [\![\Psi_1]\!]\,(w,\sigma) & \text{if } w,\sigma \models \varphi \\ [\![\Psi_2]\!]\,(w,\sigma) & \text{otherwise} \end{cases}$$

For a formula $\Psi \in$ wMSO, a given value $r \in \mathcal{R}$ can be represented by an MSO formula that describes all the strings on which Ψ returns the value r.

Definition 1. *For $\Psi \in$ wMSO and $r \in \mathcal{R}$, we define $\varphi(\Psi, r)$ recursively: $\varphi(r,r) = \top$ and $\varphi(r', r) = \neg\top$, when $r \neq r'$; and $\varphi(\psi ? \Psi_1 : \Psi_2, r) = (\psi \wedge \varphi(\Psi_1, r)) \vee (\neg\psi \wedge \varphi(\Psi_2, r))$.*

Lemma 1. *$(w,\sigma) \in [\![\phi(\Psi, r)]\!]$ iff $[\![\Psi]\!]\,(w,\sigma) = r$.*

3 Axioms

The main concern of our on-going work is to give a complete axiomatisation of wMSO. Our axiomatisation relies on an axiomatisation of MSO (or FO) on finite strings. Since satisfiability is decidable for both these logics, they have recursive and complete axiomatisations. For the case of MSO, such an axiomatisation has been given in [6], although we are not aware of a similar axiomatization for FO.

For wMSO, we first have to consider what it means to axiomatize a real-valued, non-Boolean logic. On the syntactic side, it seems natural to give an axiomatisation in terms of an equational system, denoted $\vdash \Psi_1 \approx \Psi_2$, and on the semantic side to equate two formulas that give the same value on all models, denoted $\Psi_1 \sim \Psi_2$. This also agrees with the work by Mio et al. [7] on axiomatising Riesz modal logic, which is the only other work on axiomatising real-valued logics that we know of. We augment this definition slightly by considering a set of MSO formulas Γ which we think of as assumptions. Then we write $\Gamma \vdash \Psi_1 \approx \Psi_2$ if Ψ_1 and Ψ_2 can be derived from the axioms under the assumptions Γ and $\Psi_1 \sim_\Gamma \Psi_2$ if Ψ_1 and Ψ_2 give the same values on all models that satisfy all the formulas of Γ.

We propose an axiomatization that includes the usual axioms for equality and the axioms in Table 1. The main part of the axiomatisation is concerned with axiomatising the behaviour of the conditional operator $\varphi \,?\, \Psi_1 : \Psi_2$.

Theorem 1 (Completeness of wMSO)**.** *$\Psi_1 \sim_\Gamma \Psi_2$ if and only if $\Gamma \vdash \Psi_1 \approx \Psi_2$.*

4 Further Concerns

Satisfiability For a real-valued, non-Boolean logic such as wMSO, familiar notions such as satisfiability need to be redefined, since we no longer have a satisfaction relation, but each formula is instead a function. We therefore wish to discuss and investigate how to generalise the notion of satisfiability to the real-valued setting, and determine the decidability and complexity of such notions. Plausible candidates for an extension of satisfiability is asking if a formula can return a specific value, if two formulae can return the same value, or if a formula can return a value other than a given one.

Complexity Each of these notions, as well as provability for the equational theory, can be reduced to MSO satisfiability and are thus decidable. For instance, Ψ_1 returns the same value as Ψ_2 iff

$$\bigvee_{\substack{r \text{ in } \Psi_1 \\ \text{and } \Psi_2}} \varphi(\Psi_1, r) \wedge \varphi(\Psi_2, r)$$

is satisfiable. However, one can not really hope for efficient algorithms, since the satisfiability problem for MSO on finite strings is already non-elementary [8], and MSO-satisfiability can be reduced to any of the three variants of wMSO satisfiability discussed above — for instance, φ is satisfiable iff $\varphi \mathbin{?} 1 : 0$ can take the value 1.

Another interesting theory would be the one of *inequalities*. Similarly to the above, we can see that $\Psi_1 \leq \Psi_2$ can be described by the MSO formula

$$\bigvee_{\substack{r_1 \text{ in } \Psi_1 \\ r_2 \text{ in } \Psi_2 \\ r_1 \leq r_2}} \varphi(\Psi_1, r_1) \wedge \varphi(\Psi_2, r_2),$$

and therefore, the same decidability and complexity observations can be made in this setting.

Variations Instead of using MSO formulas for conditions, one can use formulas of another logic, such as first-order logic (FO), resulting in wFO, which leads to different representation results [3]. Our complete axiomatization is agnostic with regard to the logic that one uses for conditions, as long as this logic has a complete axiomatization. One may hope to obtain better complexity results with respect to the decision problems discussed previously, by making a different choice with regards to the base logic. However, this seems unlikely in the case of FO, since, similarly to MSO, FO-satisfiability is non-elementary [8], and model checking for FO is PSPACE-complete [9, 11].

The full wMSO logic described in [5] includes a third layer called core-wMSO, which allows one to form multisets of values. This layer includes a sum over formulas indexed by a second-order variable, which corresponds to a kind of union over multisets. This sum behaves like a quantifier, so we hope that one can add axioms reminiscent of those for quantifiers of MSO in order to obtain a complete axiomatisation, although this is on-going work.

References

[1] J. Richard Büchi. Weak second-order arithmetic and finite automata. *Math. Log. Quarterly*, 6(1–6):66–92, 1960.

[2] Manfred Droste and Paul Gastin. Weighted automata and weighted logics. *Theor. Comput. Sci.*, 380(1–2):69–86, 2007.

[3] Manfred Droste and Paul Gastin. Aperiodic weighted automata and weighted first-order logic. In Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen, editors, *Proc. of 44th Int. Symp. on Mathematical Foundations of Computer Science, MFCS 2019*, volume 138 of *Leibniz Int. Proc. in Informatics*, pages 76:1–76:15. Dagstuhl Publishing, 2019.

[4] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98(1):21–51, 1961.

[5] Paul Gastin and Benjamin Monmege. A unifying survey on weighted logics and weighted automata - core weighted logic: minimal and versatile specification of quantitative properties. *Soft Comput.*, 22(4):1047–1065, 2018.

[6] Amélie Gheerbrant and Balder ten Cate. Complete axiomatizations of fragments of monadic second-order logic on finite trees. *Log. Methods Comput. Sci.*, 8(4), article 12, 2012.

[7] Matteo Mio, Robert Furber, and Radu Mardare. Riesz modal logic for Markov processes. In *Proc. of 32nd Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS 2017*, pages 1–12. IEEE Comput. Soc., 2017.

[8] Klaus Reinhardt. The complexity of translating logic to finite automata. In Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors, *Automata Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lect. Notes in Comput. Sci.*, pages 231–238. Springer, 2002.

[9] Larry J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory.* PhD thesis, MIT, 1974.

[10] Boris A. Trakhtenbrot. Finite automata and the logic of monadic predicates. *Dokl. Akad. Nauk SSSR*, 149:326–329, 1961.

[11] Moshe Y. Vardi. The complexity of relational query languages (extended abstract). In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proc. of 14th Ann. ACM Symp. on Theory of Computing, STOC '82*, pages 137–146. ACM, 1982.

Generating Test Cases Satisfying MC/DC from BDDs

Faustin Ahishakiye, Volker Stolz, and Lars Michael Kristensen

Western Norway University of Applied Sciences, Bergen, Norway
{faustin.ahishakiye,volker.stolz,lars.michael.kristensen}@hvl.no

1 Introduction

Binary decision diagrams (BDDs) are a canonical form for representing Boolean functions. BDDs have found wide application in many computer aided design (CAD) tasks [7], symbolic model checking [5, 7], verification of combinational logic [6], verification of finite-state concurrent systems [2], symbolic simulation [6], and logic synthesis and optimization. Different operations used for providing BDDs with efficient graph representation include *reduce, apply, restrict, compose* and *satisfy* [2]. In this paper, we investigate a new application area of BDDs for structural coverage analysis. We propose an approach of encoding modified condition decision coverage (MC/DC) using BDDs to obtain a minimum set of test cases. According to the definition of MC/DC [3, 9], each condition in a decision has to show an independent effect on that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions (Unique Cause (UC) MC/DC), or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome (Masking MC/DC). MC/DC is required by certification standards such as, the DO-178C [8] in the domain of avionic software systems, as a coverage criterion for safety critical software at the conditions level. It is highly recommended due to its advantages of being sensitive to program structure, requiring few test cases ($n + 1$ for n conditions), and its uniqueness due to the independence effect of each condition. If MC/DC is encoded with BDDs, a minimum set of test cases can be deduced from a reduced ordered BDD (ROBDD) based on the shortest paths. In addition, optimization and coupling conditions (for example B and $\neg B$), which are problematic for MC/DC are handled efficiently using BDDs. We present an algorithm which, given a Boolean expression in the form of an ROBDD, constructs a minimum set of test-cases satisfying MC/DC. Results show that our approach is more efficient compared to selecting MC/DC test cases from a truth table. For the rest of this paper the term BDD refers to ROBDD.

2 Minimum set of MC/DC test cases from BDDs

Theorems 1 and 2 are used to check UC MC/DC and Masking MC/DC from the BDD [1]. For both of them, there should be a pair of paths from that condition to the terminal node 1 and 0, but UC MC/DC requires strictly same path through BDD and cannot be achieved with coupling conditions. Let $f(A, B, C) = (A \wedge B) \vee C$ be a Boolean function, the truth table representing all possible MC/DC pairs is given in Table 1a. To achieve MC/DC we only need the set of $n + 1$ test cases as shown in Table 1b (one MC/DC pair for each condition).

Theorem 1. *Given a decision D, a pair of test cases of D satisfies Unique Cause MC/DC for a condition C if and only if: 1) both reach C using the same path through BDD(D); 2) their paths from C exit on two different outcomes and do not cross each other (C excluded).*

Theorem 2. *Given a decision D, a pair of test cases of D satisfies Masking MC/DC for a condition C if and only if: 1) both reach C; 2) their paths from C exit on two different outcomes and do not cross each other (C excluded).*

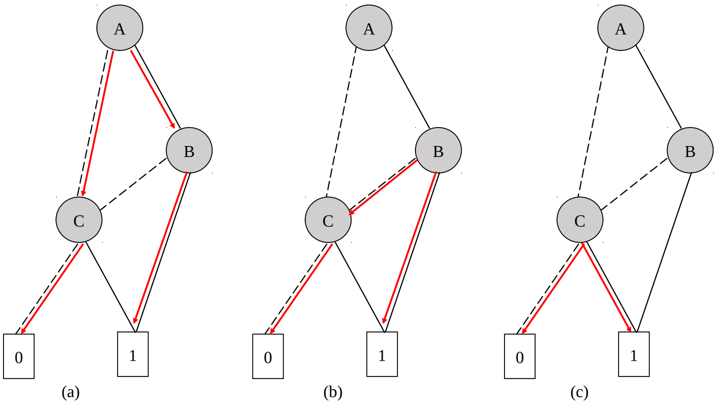

Figure 1: BDD with MC/DC minimum test cases for $(A \wedge B) \vee C$

Nr	A	B	C	f	MC/DC pairs
1	0	0	0	0	
2	0	0	1	1	C(1,2)
3	0	1	0	0	
4	0	1	1	1	C(3,4)
5	1	0	0	0	
6	1	0	1	1	C(5,6)
7	1	1	0	1	A(3,7), B(5,7)

(a) All possible MC/DC pairs

Nr	A	B	C	f	MC/DC pairs
1	0	?	0	0	
2	1	1	?	1	A(1,2)
3	1	0	0	0	B(2,3)
4	0	?	1	1	C(3,4)

(b) Required $n + 1$ MC/DC pairs

Table 1: MC/DC pairs for $f(A, B, C) = (A \wedge B) \vee C$

Some conditions may have more than one pair. For instance condition C has three MC/DC pairs, and for example if we pick C(1,2), MC/DC is achieved with five test cases ($\geq n + 1$). From the truth table it is not easy to determine which one to choose. Our idea is to take those with the shortest path in the BDD. This is useful especially for a tester, since shorter path means that the tester has to worry about less conditions. The BDD representing $f(A, B, C)$ is shown in Figure 1 (where dashed line:0/false and solid line:1/true). A test case is then simply an assignment of track-values to conditions. For an ROBDD with order $C = [A, B, C]$, $(0, ?, 1)$ denotes the test case with $A = 0$, $C = 1$, and any arbitrary value for B. If we consider the shortest paths from each node to terminal nodes 0 and 1 (shown with red arrows), the minimum set (ψ_{min}) of MC/DC test cases corresponds to $n + 1$, equivalently to the Table 1b. Note that the shorter path in the BDD corresponds to the short circuit evaluation "?" in a truth-table where the condition is not evaluated at all. We provide an algorithm that takes a BDD as input and constructs the minimum set of MC/DC test cases. It is summarized in the following steps:

1. Given an ROBDD over the list of variables C. Initialize a set of MC/DC pairs (ψ_i) and the minimum set of MC/DC pairs (ψ_{min}) as empty. The distance from each node u_i to itself is $d(u_i, u_i) = 0$, and to a terminal node is initialized as infinity ($d(u_i, v) = \infty$). The weight of each path σ as $w[\sigma] = \sum_{i=1}^{k} w(u_{i-1}, u_i)$ where the last node $u_i = v$ is the node 0 or 1. To find the shortest path, we compare the distance as $d(u_i, v) > d(u_i, u_i)+ w[\sigma]$.

2. While the list C is not empty, find the set of two shortest paths (ψ_i) from each node to 1

and 0 terminal respectively based on Theorem 1 or 2. Only new paths are added to ψ_{min}, to avoid overwriting test cases. For the example in Figure 1, $\psi_1 = \{(0,?,0),(1,1,?)\}$ for node A is added to ψ_{min}, $\psi_2 = \{(1,0,0),(1,1,?)\}$ for node B, only $(1,0,0)$ will be added to ψ_{min}, and $\psi_3 = \{(0,?,0),(0,?,1)\}$ for node C and only $(0,?,1)$ is added to ψ_{min}.

3. If the list is empty, the algorithm returns $\psi_{min} = \{(0,?,0),(1,1,?),(1,0,0),(0,?,1)\}$ which is equivalent to $n+1$ required MC/DC test cases as shown in Table 1. Otherwise, MC/DC can not be achieved. The uncovered conditions will be the remaining variables on C.

In case of strongly coupled conditions (for example B and $\neg B$), our algorithm can still find the minimum set satisfying Masking MC/DC. In addition, BDDs implement directly the optimization for conditions in the decision. For example, $f(A,B,C){=}(A \vee \neg A \vee B \vee C) \wedge A$, its BDD representation will contain only one condition (A) because conditions B and C will never be evaluated at all and therefore, they do not appear in the BDD. The optimized representation of BDD helps testers to observe conditions with no effect to the outcome so that there is no need for checking MC/DC for those conditions.

3 Conclusion

This paper presents a new approach of generating test cases satisfying MC/DC from BDDs and provides a new algorithm to deduce the minimum set of test cases satisfying MC/DC based on the shortest path in the BDD. Our results based on small examples show that $n+1$ test cases required to achieve MC/DC for n conditions can be found from the BDD with less overhead compared to selecting them from the truth table. The algorithm can be implemented in Python based on Pyeda BDD library [4]. Moreover, we investigated the formulation of unique cause and masking MC/DC in terms of BDDs. More example results (considering coupling condition and optimization) with the implementation and formal proof of correctness for the algorithm will be provided in an extended paper.

References

[1] Adacore. Technical report on OBC/MCDC properties. Technical report, Couverture project, 2010.

[2] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[3] John Joseph Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Softw. Eng. J.*, 9(5):193–200, 1994.

[4] Drake Christopher R. Pyeda: Data structures and algorithms for electronic design automation. In *14th Python in Science Conference (SCIPY)*, 2015.

[5] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[6] Alan J. Hu. Formal hardware verification with BDDs: An introduction. In *Proc. of IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, PACRIM '97*, volume 2, pages 677–682. IEEE, 1997.

[7] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*, 1st ed. Springer, 1998.

[8] Frederic Pothon. DO-178C/ED-12C versus DO-178B/ED-12B: Changes and improvements. Technical report, AdaCore, 2012. Available at https://www.adacore.com/books/do-178c-vs-do-178b.

[9] Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013.

Back to Direct Style for Delimited Continuations*

Dariusz Biernacki, Mateusz Pyzik, and Filip Sieczkowski

Inst. of Computer Science, University of Wrocław, Wrocław, Poland
{dabi,matp,efes}@cs.uni.wroc.pl

Abstract

We present an inverse to a CPS transformation for a functional programming language with delimited control operators shift and reset, extending an earlier line of work for simpler calculi. We also study the evaluation behaviour of corresponding programs in direct and continuation-passing styles.

1 Background and overview

Continuation Passing Style (CPS), dating back to Fischer and Plotkin [8, 10], is a format for functional programs where all intermediate results are named, all calls are tail-calls and programs are evaluation-order independent [1]. In a program that adheres to CPS, all procedures take an additional argument, a *continuation*, that represents "the rest of the computation" [2]. CPS may be used, even unknowingly, as a style of hand-written code (e.g., ubiquitous callback arguments in JavaScript) or as an intermediate language in an optimising compiler (e.g., Standard ML of New Jersey). Usage of CPS in compilers would not be possible without a mechanical *CPS transformation*. Such transformation takes an arbitrary program and produces an equivalent program in CPS.

A natural question arises: can we recover the original, *Direct Style (DS)* program from its CPS-compliant derivative? The answer, established by Danvy and extended in joint work with Lawall, is affirmative: one can indeed apply a *DS transformation* to a program in CPS and obtain its DS counterpart [2, 5] when continuations are pure or abortive. In pure CPS, the continuation parameter is called exactly once, in a tail position somewhere inside the procedure body. As a rule of a thumb, all deviations from this pattern should warn us that the control flow in the procedure is nonstandard, and that the author is trying to employ some computational effect: exceptions, Prolog-like backtracking, coroutines etc. In [5], Danvy and Lawall show that we can render some nonstandard uses of continuations (i.e. abortive) in Direct Style by using the `call-with-current-continuation` (or `call/cc` for short) *control operator* [11].

The `call/cc` operator is a prime example of an undelimited (or, *abortive*) control operator: once called, the continuation never returns to its call site – in effect, modelling a jump. In this paradigm, a continuation spans the entire future of the program's execution. However, some computational effects do not behave like jumps: one vivid example is backtracking. In this scenario, continuations model only a fragment of the future program execution, i.e., they are *delimited*, and we use them in non-tail positions, making them *composable*. In direct style, this behaviour is embodied by the control operators `shift` and `reset` whose image under the CPS translation gives rise to composable continuations [4]. The `shift` operator captures the current continuation but – in contrast to the more common `call/cc` – one that extends only up to the dynamically nearest enclosing `reset` operator. Interestingly, these operators have the capability to express *any* computational effect [7] and are closely related to the recent idea of *algebraic effects* [9]. However, in contrast to pure lambda calculus and abortive continuations,

*This work was partially supported by National Science Centre, Poland, grant no. 2014/15/B/ST6/.

the question of a direct-style transformation for a language with composable continuations has not yet been studied.

In this work we attempt to establish such a correspondence. To this end we first introduce a revised CPS transformation for a functional language with delimited control operators `shift` and `reset` that preserves all language redexes. We characterise the image of the translation by an inductive judgement, extending the approach used by Danvy, Lawall and Pfenning [3, 5, 6]. We define an effective Direct Style transformation directed by this judgement and prove that the two transformations are mutual inverses. Moreover, we study the evaluation behaviour of corresponding DS and CPS programs, following the work on Galois reflections for pure call-by-value lambda calculus by Sabry and Wadler [12]. We find that together, CPS and DS transformations establish an isomorphism between Direct Style language and the image of the CPS transformation, where CPS terms are equipped with a custom evalution relation. We are currently working on extending this result to general reduction.

2 Technical details

Direct Style language Our DS language λ_{SRT} is λ_v enriched with delimited control operators `shift`/`reset` together with somewhat artificial but later justified `throw` operator. We distinguish continuations from ordinary functions, i.e. they are not first-class. Continuations are applied using a dedicated `throw` syntax.

CPS transformation Our CPS transformation comes in equivalent second- and first-order formulations which we use interchangeably in proof developments as needed. We follow the general ideas of the *Back to direct style* papers [3, 5], devising inductive judgements which are meant to be provable if and only if a judged expression is in the range of CPS transformation. Moreover, there is at most one matching proof for each expression. To achieve such uniqueness, our CPS is sprinkled with *explicit redexes* which mark the occurences of control operators in CPS.

DS transformation The judgements are designed so that one can effectively recover the corresponding direct style term from a proof-term. In order to devise an effective and not only a theoretical inverse transformation one needs to recover the proof-term from a CPS expression. Such a procedure is neccessarily partial, as not all lambda expressions are in CPS. Most of the cases can be easily distinguished by a shallow inspection of the expression but special care needs to be taken with `shift` and `reset`. These two are hard to differentiate if not for a one peculiarity: serious expressions have a different parity of outermost λ-abstractions than continuation expressions. This subtle difference saves us from trouble and is the primary reason for keeping continuations second-class. Otherwise, distinction would disappear and it would be possible to disprove uniqueness of proof-terms. Uniqueness is crucial here – without it an inverse transformation becomes multi-valued, i.e., many different DS expression might give the same CPS expression. We do not have a decisive answer whether this problem with first-class continuations can be overcome by a different CPS transformation.

Evaluation in CPS Seemingly redundant explicit redexes appear whenever we translate a control operator. They allow us to clearly distinguish between operators before and after evaluation step. This ensures that each step of evaluation in a DS expression shall be mirrored in a CPS transform and *vice versa*. In order to establish a genuine lockstep between these, we aggregate some β-contractions in the CPS language, making it illegal to stop evaluation in certain moments. We refine the operational semantics in the range of the CPS transformation in order to achieve monotonicity properties; they do not seem to hold when *vanilla* λ_v evaluation is considered.

We define a subrelation of ordinary λ_v reduction on the range of the CPS transformation. Usual β-reductions are allowed only in specific syntactic positions which correspond to application and control operators. This allows us to move along evaluation in CPS just as if we evaluated in Direct Style and performed CPS transformation each time. In a sense, evaluation and transformation commute, or in terms of partial orders, CPS transformation is monotone. We conjecture it should be monotone when appropriate *reduction* relations are considered but this is a topic of an ongoing work.

References

[1] D. Biernacki, O. Danvy, and C. Shan. On the static and dynamic extents of delimited continuations. *Sci. Comput. Program.*, 60(3):274–297, 2006.

[2] O. Danvy. Back to direct style. In B. Krieg-Brückner, editor, *Proceedings of the Fourth European Symposium on Programming*, volume 582 in *Lecture Notes in Computer Science*, pages 130–150. Springer, 1992.

[3] O. Danvy. Back to direct style. *Sci. Comput. Program.*, 22(3):183–195, 1994.

[4] O. Danvy and A. Filinski. Abstracting control. In M. Wand, editor, *Proc. of 1990 ACM Conf. on Lisp and Functional Programming*, pages 151–160. ACM Press, 1990.

[5] O. Danvy and J. L. Lawall. Back to direct style II: First-class continuations. In W. Clinger, editor, *Proc. of 1992 ACM Conf. on Lisp and Functional Programming*, pages 299–310. ACM Press, 1992. Also in *LISP Pointers*, 5(1):299–310.

[6] O. Danvy and F. Pfenning. The occurrence of continuation parameters in CPS terms. Technical report CMU-CS-95-121, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, Feb. 1995.

[7] A. Filinski. Representing monads. In H.-J. Boehm, editor, *Proc. of 21st Ann. ACM Symp. on Principles of Programming Languages*, pages 446–457. ACM Press, 1994.

[8] M. J. Fischer. Lambda-calculus schemata. In *Proc. of ACM Conf. on Proving Assertions about Programs*, pages 104–109. ACM, 1972. Also in *ACM SIGPLAN Not.*, 7(1):104–109, 1972. Reprinted in *Lisp Symb. Comput.*, 6(3–4):259–288, 1993.

[9] Y. Forster, O. Kammar, S. Lindley, and M. Pretnar. On the expressive power of user-defined effects: effect handlers, monadic reflection, delimited control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, 2017.

[10] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theor. Comput. Sci.*, 1:125–159, 1975.

[11] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proc. of 25th ACM National Conf.*, pages 717–740. ACM, 1972. Reprinted in *Higher-Order Symb. Comput.*, 11(4):363–397, 1998.

[12] A. Sabry and P. Wadler. A reflection on call-by-value. *ACM Trans. Program. Lang. Syst.*, 19(6):916–941, 1997.

Towards Automatic Verification of C Programs with Heap

Zafer Esen and Philipp Rümmer

Dept. of IT, Uppsala University, Sweden; {zafar.esen,philipp.ruemmer}@it.uu.se

Abstract

Programs containing dynamic data structures pose a challenge for static verification, because the shape of data-structures has to be reconstructed, and invariants involving unboundedly many heap locations have to be found. In previous work in the context of the Java model checker JayHorn, a simple Horn encoding through object invariants has been proposed to model programs with heap. This abstract presents ongoing work on TriCera, a model checker for C programs with a similar representation of heap interactions.

1 Introduction

Effective handling of heap-allocated data-structures is one of the main challenges in automatic verification approaches such as software model checking. In order to verify programs operating on such data-structures, a verification tool has to analyse the *shape* of the data-structures (which is usually not explicitly expressed in a program), and also has to find *data invariants* that cover an unbounded number of heap locations. A plethora of approaches to address this challenge has been developed over the years; for instance, separation logic [5] provides a general verification methodology, but has mostly been successful in interactive verification systems. In software model checkers, the most common solution is to use a low-level representation of heap as an array; this necessitates quantified invariants to verify programs with unbounded data-structures, which in itself is a challenging research problem.

A different methodology, inspired by *refinement type systems* [2] and based on instance invariants associated with the various classes in a program, is used in the Java model checker JayHorn [4]. The model checker *TriCera*[1], presented in this abstract, applies a similar heap representation to verify C programs, but extends the method to handle also language features not present in Java: among others, pointers to stack-allocated data, and primitive heap-allocated data like integers.

The architecture of TriCera is given in Figure 1. TriCera encodes all programs as sets of Horn clauses, and then uses Eldarica [3] as its backend to try and solve these. This means that state invariants, function contracts, and object invariants are all computed automatically by the Horn solver; in the end, making the whole process fully automatic.

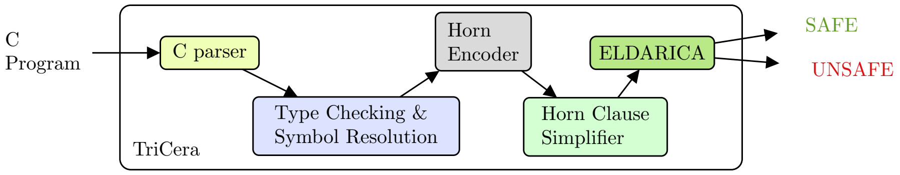

Figure 1: TriCera Architecture

[1] https://github.com/uuverifiers/tricera

```
typedef struct S1 {int f;} S1;
typedef struct S2 {int f1, f2;} S2;

S1 *s1_1 = calloc(sizeof(S1));
S1 *s1_2 = calloc(sizeof(S1));
S2 *s2   = calloc(sizeof(S2));
s1_1->f  = 42;       //push operation
int t    = s1_1->f; //pull operation

assert(t == 0 || t == 42);
```

Figure 2: A simple C program.

```
typedef struct S1 {int f;} S1;
typedef struct S2 {int f1, f2;} S2;

S1 *s1_1 = H++;     //allocation S1
push(s1_1, S1(0)); //push from calloc
S1 *s1_2 = H++;     //allocation S1
push(s1_2, S1(0)); //push from calloc/
S2 *s2 = H++;       //allocation S2
push(s2, S2(0,0)); //push from calloc
push(s1_1, S1(42); //push from assignment
S1 pulled = pull(s1_1);
int t = pulled.f

assert(t == 0 || t == 42);
```

Figure 3: A simple C program where heap interactions are replaced with **push** and **pull** operations.

2 Heap Encoding Using Invariants

Instead of modeling each data item precisely, a *heap invariant* (ϕ_{Type}) is used to represent each data type on the heap. These invariants capture the properties of the data type they correspond to. They are symbolic place-holders, which are later computed automatically by the Horn solver. Interactions with the heap are done via *push* and *pull* operations which use these invariants.

Figure 2 shows a very simplistic C program, and Figure 3 shows its reduced version where heap related operations are automatically replaced with `push` and `pull` operations. These operations are then reduced into `assert` and `assume` statements using the replacement rules given below. The final translation into Horn clauses then follows, which is straightforward.

$$y = \textbf{pull}(x) \rightsquigarrow \{\textbf{assume}(\phi_{Type}(ptr, x_{fresh})); \quad y = x_{fresh};\}$$
$$\textbf{push}(ptr, val) \rightsquigarrow \textbf{assert}(\phi_{Type}(ptr, val))$$

C `struct`s are encoded using the theory of algebraic data types (ADTs), meaning that they are represented by a single value on the heap similarly to primitive data types.

Figure 4 shows how the heap would look like for the simple program given in Figure 2, and Figure 5 shows how the heap is encoded using invariants. The properties of the objects of same type are captured by the same invariant, as in the case of `S1`.

Memory allocation, in the example using `calloc`[2], is done by assigning the value of the heap counter `H` to the pointer variable, and then incrementing the value of the counter. A zero initialized value is also *pushed* to that location in the case of `calloc`.

Assigning to a variable on the heap can be seen as updating its property, thus the heap invariant must now satisfy the new property as well. This means that the push operation *asserts* the heap invariant using the newly assigned value. On the other hand, reading from the heap can be done by creating a fresh variable which satisfies the properties associated with that type, by *assuming* that the heap invariant holds with this new variable as its argument.

3 Experiments and Results

A total of 114 benchmarks were used to evaluate the initial performance of TriCera. The benchmarks were chosen from files located under the *ReachSafety-Heap* and *MemSafety-Heap* categories of SVCOMP'19[3], and which did not contain unsupported constructs such as arrays.

[2] This differs from the standard C `calloc` function by having no argument for the number of items, as arrays are currently not supported in TriCera.

[3] https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp19

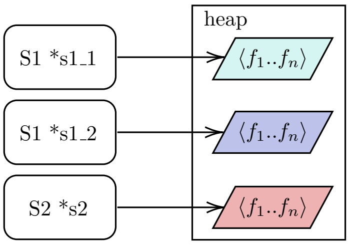
heap
S1 *s1_1
S1 *s1_2
S2 *s2
$\langle f_1..f_n \rangle$
$\langle f_1..f_n \rangle$
$\langle f_1..f_n \rangle$

Figure 4: Heap representation as a partial function which maps locations to values

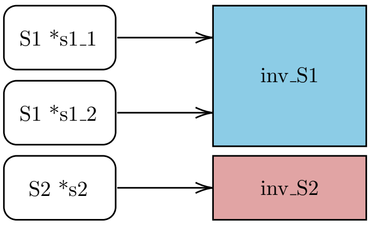

Figure 5: Heap invariants which are created for each type

The benchmark programs are provided as inputs to both TriCera (v0.1) and CPAchecker (v1.8), using the default settings of the tools.

With a timeout of 5 minutes, TriCera could verify 25 (8 safe and 17 unsafe) out of 114 programs in 11 minutes. The rest of the programs were flagged as unsafe due to the imprecision of the current heap encoding. For comparison, CPAchecker [1] could verify 53 (40 safe and 13 unsafe) out of 114 programs in 183 minutes. Both tools produced no unsound results.

While TriCera could verify correctly almost half of what CPAchecker could in total, it took one tenth of the time to do so. However, this was mostly due to CPAchecker timing out trying to verify tasks, on which TriCera gave up much earlier and produced false alarms. It is expected that the refinements discussed in Section 4 should reduce the number of these false alarms, while keeping a similar level of performance with respect to execution time.

4 Conclusions and Future Work

This paper presented the ongoing work with TriCera. The initial results are promising; however, there are several planned improvements to increase the precision, such as using *allocation sites* as done in JayHorn [4] and adding flow sensitivity. There are also plans to to support a wider subset of the C language, such as arrays and pointer arithmetic.

Since the whole encoding is over-approximate, one cannot directly trust the generated counterexamples. To get a genuine counterexample, the encoding can also be complemented with an under-approximate encoding, as done in JayHorn.

References

[1] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proc. of 23rd Int. Conf. on Computer Aided Verification, CAV 2011*, volume 6806 of *Lect. Notes in Comput. Sci.*, pages 184–190. Springer, 2011.

[2] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. of ACM SIGPLAN 1991 Conf. on Program. Language Design and Implementation, PLDI '91*, pages 268–277. ACM, 1991.

[3] Hossein Hojjat and Philipp Rümmer. The ELDARICA Horn solver. In *18th Int. Conf. on Formal Methods in Computer Aided Design, FMCAD 2018*, 7 pp. IEEE, 2018.

[4] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. Jayhorn: A framework for verifying java programs. In Swarat Chauduri and Azadeh Farzan, editors, *Proc. of 28th Int. Conf. on Computer Aided Verification, CAV 2016, Part I*, volume 9779 of *Lect. Notes in Comput. Sci.*, pages 352–358. Springer, 2016.

[5] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of 17th IEEE Symp. on Logic in Computer Science, LICS 2002*, pages 55–74. IEEE, 2002.

Ready, Set, Go!
Data Race Detection and the Go Language

Daniel Schnetzer Fava and Martin Steffen

Department of Informatics, University of Oslo, Norway
{danielsf,msteffen}@ifi.uio.no

Abstract

When compared to other synchronization methods (such as locks), channels communication has arguably received lesser attention from the data race checking community. In this talk, we will present a novel data race checker for a concurrent language featuring channel communication as its sole synchronization primitive.

Background

Early on, the concept of race, meaning the competition for access to a shared resource, was explored in the context of channels [12]. Memory models were absent in that setting and, instead, agents were said to race when sending or receiving messages from the same channel "at the same time." Race checking was then conflated often with confluence or determinacy checking [1], with different schedules leading to potential variations in the flow of information through a system.

Today, more often than not, the word *race* is used to denote a *data race*, where threads compete when performing read and write accesses to the same location in shared memory. Because locks have become a very common mechanism for ensuring race-freedom, the contemporary discussion on data race checking is often devoid of the notion of channels; neither does confluence tend to enter the picture. A notable exception is Terauchi and Aiken [17], who give a modern treatment of confluence checking in the context of channel communication. Their work incorporates a form of shared memory, which the authors call *cell channels*. Their treatment, however, does not address data race checking as a first class property but only as a consequence of determinacy. Although related, data-race freedom is different from determinacy and confluence. There exist non-confluent scenarios that are properly synchronized. It is interesting to note, however, that the task of data race checking in the context of channels has received little attention.

Even though mutual-exclusion has dominated the synchronization landscape, languages based on message passing do have a strong footing. Take the Go programming language as an example [7, 2]. Go has gained traction in networking applications, web servers, distributed software and the like. "It prominently features goroutines, which are asynchronous functions resembling lightweight threads, and buffered channel communication in the tradition of CSP [9] (resp. the π-calculus [13]) or Occam [10]. While encouraging message passing as the prime mechanism for communication and synchronization, threads can still exchange data via shared variables. Go's specification includes a memory model [5] which spells out, in precise but informal English, the few rules governing memory interaction at the language level" [3].

Our approach

In prior work, we have given an operational formalization of a relaxed memory model inspired by the Go memory model's specification. We also proved what is called the DRF-SC guarantee

[3], meaning that the memory model behaves *sequentially consistently* when executing data-race free programs. We now propose a data race checker based on our relaxed memory model and the DRF-SC result in particular.

$$\dfrac{(m, !z) \in E_{hb} \qquad E^r_{hb} \subseteq E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = \{(m', !z)\} \cup (E_{hb} - E_{hb} \downarrow_z)}{p\langle E_{hb}, z := v'; t\rangle \parallel m\llparenthesis E^r_{hb},\ z{:=}v\rrparenthesis \rightarrow p\langle E'_{hb}, t\rangle \parallel m'\llparenthesis \emptyset,\ z{:=}v'\rrparenthesis}\ \text{R-Write}$$

$$\dfrac{\begin{array}{c} E'^r_{hb} = \{(m', ?z)\} \cup (E^r_{hb} - E_{hb} \downarrow_z) \\ (m, !z) \in E_{hb} \qquad \mathit{fresh}(m') \qquad E'_{hb} = \{(m', ?z)\} \cup (E_{hb} - E_{hb} \downarrow_z) \cup \{(m, !z)\} \end{array}}{p\langle E_{hb}, \texttt{let}\ r = \texttt{load}\ z\ \texttt{in}\ t\rangle \parallel m\llparenthesis E^r_{hb},\ z{:=}v\rrparenthesis \rightarrow p\langle E'_{hb}, \texttt{let}\ r =\ v\ \texttt{in}\ t\rangle \parallel m\llparenthesis E'^r_{hb},\ z{:=}v\rrparenthesis}\ \text{R-Read}$$

Figure 1: Strong operational semantics augmented for data-race detection

The detector works by recording read- $(m, ?z)$ and write-events $(m', !z)$, where m, m', etc. are unique identifiers. These events are recorded along a variable's memory cells and in thread-local storage. The memory cell associated with a shared variable z takes the form $m\llparenthesis E^r_{hb},\ z{:=}v\rrparenthesis$ where m is the identifier of the most recent write to z, E^r_{hb} is a set holding read events for loads that have accumulated since the most recent write to z, and v is the variable's current value. Threads are equipped with a set E_{hb} which holds information about read- and write-events that are "known" to the thread as having *happened-before*. When a thread attempts to access a memory location, the detector checks whether there exists a concurrent memory event that conflicts with the attempted access—rules R-Write and R-Read of Figure 1. Not shown here are the rules regarding channel communication. Communication carries over happens-before information between threads, thereby affecting synchronization.

In this talk, we will show that the information needed for race checking is contained in the scaffolding of the DRF-SC guarantee proof. Given our experience, we conjecture that, in general, race checkers may be (semi-)automatically derivable from memory models and their corresponding DRF-SC proofs. We should point out, however, that the operational semantics we propose for data race detection is *not* a weak semantics. Apart from the additional bookkeeping, the semantics is "strong" in that it formalizes a memory guaranteeing sequential consistency. Note that, to focus on a form of strong memory is not a limitation. Given that even racy program behaves sequentially consistently up to the point in which the first data-race is encountered, a complete race detector can safely operate under the assumption of sequential consistency.

When it comes to channel communication, our treatment focuses on bounded channels, including synchronous ones. While channels are often used to enforce order between events from different threads (e.g. a send happens before the corresponding receive completes), we revisit the link between channels and locks and discuss how the effects of a channel's boundedness can be used to ensure mutual-exclusion. The pervasiveness of the notion of mutual-exclusion might explain why it is often thought that a data race involves two (or more threads) accessing memory *at the same time*—or, if not at the same time, that there exists an alternate yet equivalent run in which the conflicting accesses can be placed side-by-side. We argue that this interpretation is misleading. Specifically, we show that there exist race conditions in which conflicting accesses are necessarily ordered and necessarily separated in time by other operations. This observation has interesting and subtle consequences relating back to *anomalies* discussed by Lamport in his seminal paper on vector-clocks and distributed systems [11]. In the talk, we will discuss anomalies arising from the fact that the first-in-first-out nature of channels is not fully accounted for in the happens-before relation as defined by the Go memory model's description. We show alternatives for handling the anomalies, but settle with a race detector formalization that is faithful to the Go language specification.

Finally, we compare our approach to modern data race checkers based on vector clocks. In particular, we contrast our work against that of Pozniansky and Schuster, which introduced an algorithm sometimes referred to as Djit$^+$ [14], and to Flanagan and Freund and their work on FastTrack [4]. Such algorithms have influenced the Thread Sanitizer library [15, 16, 8] upon which the current Go race detector [6] is implemented.

References

[1] Robert Cypher and Eric Leu. Efficient race detection for message-passing programs with nonblocking sends and receives. In *Proc. of 7th IEEE Symp. on Parallel and Distributed Processing, SPDP '95*, pages 534–541. IEEE, 1995.

[2] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015.

[3] Daniel S. Fava, Martin Steffen, and Volker Stolz. Operational semantics of a weak memory model with channel synchronization. *J. Log. Algebr. Methods Program.*, 103:1–30, 2019.

[4] Cormac Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In Michael Hind and Amer Diwan, editors, *Proc. of 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '09*, pages 121–133. ACM, 2009.

[5] Go language memory model. `https://golang.org/ref/mem`, 2014. Version of May 31, 2014, covering Go version 1.9.1.

[6] Go language race detector. `https://blog.golang.org/race-detector`, 2013.

[7] Go language specification. `https://golang.org/ref/spec`, 2016.

[8] Google Thread Sanitizer library.
`https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm`, 2015.

[9] Charles A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[10] Geraint Jones and Michael Goldsmith. *Programming in Occam2*. Prentice-Hall, 1988.

[11] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[12] Friedemann Mattern. Virtual time and global states in distributed systems. In Michel Cosnard et al., editors, *Proc. of Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier, 1988.

[13] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Inf. Comput.*, 100(1):1–40, 41–77, 1992.

[14] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proc. of 9th ACM Symp. on Principles and Practice of Parallel Programming, PPoPP '03*, pages 179–190. ACM, 2003.

[15] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proc. of Workshop on Binary Instrumentation and Applications*, pages 62–71. ACM, 2009.

[16] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with LLVM compiler: compile-time instrumentation for ThreadSanitizer. In Safraz Khurshid and Koushik Sen, editors, *Revised Selected Papers from 2nd Int. Conf. on Runtime Verification, RV 2011*, volume 7186 of *Lecture Notes in Computer Science*, pages 110–114. Springer, 2012.

[17] Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Program. Lang. Syst.*, 30(5), article 27, 2008.

Futures, Histories and Smart Contracts

Elahe Fazeldehkordi and Olaf Owe

Department of Informatics, University of Oslo, Norway
{elahefa,olaf}@ifi.uio.no

Abstract

In this work we reconsider the concept of shared futures as used in the setting of asynchronous method-oriented communication. The future mechanism offers some advantages such as decoupling of method invocation from retrieval of the result, and sharing of the result. However, it also has some drawbacks, including the need of garbage collection and poor privacy protection. We here suggest a more general and more expressive notion than futures, but without the two mentioned drawbacks. We use the term *history objects* for this new concept since it encapsulates past communications in addition to the information normally found in futures. We argue that the suggested new concept has several advantages, including support of functionalities found in smart contracts.

Keywords: Futures; Transactions; Asynchronous Communication; Smart Contracts.

Introduction

We reconsider the future concept, which has become popular in the setting of concurrent objects (or agents) communicating asynchronously. This setting is adopted in the active object paradigm, supported by several languages [2]. Remote method calls are handled by message passing and the result of a method invocation is placed in a future object, at which time the future is said to resolved. The caller generates a reference to the future object and this reference may be passed to other objects. Any object with a reference to the future object may ask for the value, typically via a *get* statement, which will block when the future is not yet resolved (some languages allow polling to check if a future is resolved). The main advantages of the future mechanism are improved flexibility compared to the traditional blocking remote call mechanism, delaying or avoiding the blocking, and providing safe sharing of results since a future is a write-once, multiple-read data store. Without polling the future mechanism is race-free, since a get operation waits while the future value is not there. Polling, would make an object sensitive to the speed of object in the environment (something which is often acceptable).

However, it is not trivial to detect when a future can be discarded; and as many futures may be generated, garbage collection is in general needed. In the active object paradigm, this is a clear disadvantage since the active objects themselves have a long life time. If local data inside objects is defined by data types, using a functional programming language to express and manipulate values of the data type (such as in the Creol and ABS languages), there is no need for general garbage collection of these values, assuming storage for values of the data types can be retrieved efficiently. Another disadvantage of the future mechanism is that the future value is unprotected, and an object getting the value may not know where the future came from and what it represents. In particular, privacy aspects are unknown and the information can easily be misused [4].

In this work we propose a new language construct to reduce these drawbacks. We propose a "container box" for recording all calls and futures related to the interactions involving a given object. This "container" will then hold all future values generated by the given object in the same "box", including present and past communications. For this reason we will call it *history object.* For simplicity we consider one history object for one active object, but one could associate several history objects with one object, reflecting the interactions according to different interfaces of the object. The aim of this abstract is to sketch and motivate this new language primitive.

History objects

Notation: We consider a syntax similar to that of the Creol/ABS language family [3]. We let v denote a variable, x a formal parameter (assumed to be read-only), o an object variable (an object reference), f a future variable (a reference to a future object), e an expression (assumed to be pure), and m a method. We use capitalized words for types and interfaces, while variable and method names start with a lower-case character. We use [...] for lists and < ... > for tuples. The statement syntax v: + e is permitted when v is a list to express that an element e is appended to the tail of list v. This statement adheres to the *write-once* discipline since it cannot change previously written elements.

As is the case in the traditional future concept, we consider three kinds of method calls:

- A *blocking call* has the syntax $v := o.m(\overline{e})$, where o is the callee, m the called method, $\overline{e}$ the (input) parameters. The method result will be assigned to v.
- A *simple asynchronous call* has the syntax $o!\,m(\overline{e})$. The caller is not blocked and the result value is not communicated to the caller.
- An *asynchronous call* has the syntax $f := o!\,m(\overline{e})$ where f is a future variable declared with type $Fut[T]$ where T is the return type of m. The variable f is assigned a new call identity (a reference to the future object) uniquely identifying the call. This identity may be communicated to other objects. The caller is not blocked. In order to obtain the value returned from the call, the caller object (or another object that knows the future identity) may perform $v :=$ **get** f where v is a program variable of type T. This statement will block if the future is not resolved, and otherwise the result value is copied into v.

Consider now our proposed setting (a bit simplified) where we associate a history object, $history(o)$, to each object o called with an asynchronous call. We allow the same call operations as above. But there are two major differences: In our proposed solution, the same history object contains all future values generated by a given callee object, including those of the past as well as future ones. This history object is therefore long-lived and need not be garbage-collected. However, as the storage need is growing dynamically, the history objects could be placed in a cloud. The history object could be split over several objects if needed. (In that case for each function definition in the top level history object, the function value over the empty history must correspond to the function value over the final history in the underlying object.)

Secondly, we treat the history objects as normal objects, which means that one may communicate with the history objects when desired, using simple or blocking calls. This is following the spirit of [5]. Moreover, the behavior of the history objects is given by interface declarations, including a *get* method corresponding to reading a future value. A predefined class implementation can be given, and by means of inheritance, this class declaration may be extended and modified in the same way as other classes. In particular one may add functionality by implementing new interfaces and methods, and one may add protection mechanisms in redefined *get* methods to implement security and privacy restrictions. For instance by requiring that the caller satisfies some conditions. In the case of a two-party contract, we can require that only the two parties may see the history (through *get* calls). This means that calls to the history objects may distinguish their behavior depending on the caller (using the implicit parameter caller and its interface).

The predefined history object contains a (private) transaction list

```
List[Transaction] trans = empty
          // restricted by incremental-write/multiple-read access
```

and a predefined (hidden) *put* method to be used only by the underlying runtime system for recording each new message to or from the object by appending it to the transaction list.

```
Void put(Transaction t) {trans :+ t} //appending t to trans
```

We allow read access to the transaction list. The transaction corresponding to a call $f := o!\,m(\overline{e})$ is a tuple of form

$$< fid, caller, method, par, result >$$

where fid is a future identity (the value assigned to f), *caller* is the caller identity, *method* is the method name (m), *par* is the input values (the values of $\overline{e}$), and *result* is the result of the call, possibly *error*. This transaction is generated when the call has completed normally or abnormally (i.e., resulted in an error). The statement **return** e in the body of a method $m(\overline{T\ x})$ is executed at runtime by doing $history(this).put(< callid, caller, "m", [\overline{x}], e >)$ where this refers to the current object, $\overline{x}$ are the formal parameters, and where *callid* and *caller* are implicit parameters in our setting giving the future identity of the call and the caller object, respectively. (The *put* statements do not appear in the program code.) Abnormal termination results in a *put* call with result *error*. Similarly, a history object includes a public *get* method T **get**$(Fut[T]\ f)$ for each method result of type T. A get statement is therefore possible in our setting as a blocking call to *get* on the appropriate history object.

We observe that the transaction history of an active object as given by its history object, is sufficient to define the state of the object at the end of a method execution. The state of an active object at its last method completion can be reconstructed from the transaction history, and also the pre-state of method execution resulting in error. (Cooperative scheduling would require more events to be recorded in the *trans* variables.) We also observe that the history objects define the transaction history in a faithful way due to the language restriction on the *trans* variables by means of incremental-write by the underlying runtime system and is immutable for others. This restriction is also imposed on subclasses of the history classes. This means that one may rely on the transaction histories in a way similar to smart contracts. This is checked statically. If the runtime system also offers protection of unauthorized write access to these variables by means of a trusted execution environment, one does not need block-chain technology to guarantee incremental-write/multiple-read access. If not, one may use an underlying block-chain technology at runtime to obtain full trust. An active object with an associated history object may be used to accommodate the functionality of a smart contract, and with immutability of the transaction history guaranteed at the software level. We support the following aspects of smart contracts:

- *Trust* is provided by the reliable independent history object.
- *Reliability* due to the write-once (by the underlying operating system) and multiple-read access by users.
- A way to *formally specify and verify contract requirements* as well as invariants using the communication history and defining functions over the history.
- *Roll-back* possibility in case a transaction cannot complete.

Further details and examples will be given in an extended version of this paper, where we will also consider an example of a smart contract for auctions [1].

References

[1] W. Ahrendt, G. J. Pace, and G. Schneider. Smart contracts: A killer application for deductive source code verification. In P. Müller and I. Schaefer, editors, *Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of His 60th Birthday*, pages 1–18. Springer, 2018.

[2] F. de Boer and et al. A survey of active object languages. *ACM Comput. Surv.*, 50(5):1–39, 2017.

[3] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. K. Aichernig. F. S. de Boer, and M. Bonsangue, editors, *Revised Papers from 9th Int. Symp. on Formal Methods for Components and Objects, FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2012.

[4] F. Karami, O. Owe, and T. Ramezanifarkhani. An evaluation of interaction paradigms for active objects. *J. Log. Algebr. Methods Program.*, 103:154–183, 2019.

[5] I. Sergey and A. Hobor. A concurrent perspective on smart contracts. arXiv preprint 1702.05511, 2017.

Marking Piecewise Observable Purity

Seyed Hossein Haeri[*] and Peter Van Roy

INGI, Université catholique de Louvain, Belgium; {hossein.haeri,peter.vanroy}@uclouvain.be

Abstract

We present a calculus for distributed programming with explicit node annotation, built-in ports, and pure blocks. To the best of our knowledge, this is the first calculus for distributed programming, in which the effects of certain code blocks can go unobserved, making those blocks pure. This is to promote a new paradigm for distributed programming, in which the programmer strives to move as much code as possible into pure blocks. The purity of such blocks wins familiar benefits of pure functional programming.

1 Introduction

Reasoning about distributed programs is difficult. That is a well-known fact. That difficulty is partly because of the relatively little programming languages (PL) research on the topic. But, it also is partly due to the unnecessary complexity conveyed by the traditional conception of distributed systems.

Reasoning about sequential programs written for a single machine, in contrast, is the bread and butter of the PL research. Many models of functional programming, for example, have received considerable and successful attention from the PL community. The mathematically rich nature of those models is the main motivation.

In particular, pure functional programming, i.e., programming with no side-effects, has been a major source of attraction to the PL community. In that setting, many interesting properties are exhibited elegantly, e.g., equational reasoning, referential transparency, and idempotence.

The traditional conception about distributed programming, however, lacks those nice properties. In that conception, side-effects are inherent in distributed programming. The argument is: No sensible distributed program can run without communication between nodes; and, such communications **always** alter the state of the communication medium; hence, the effectfulness.

The *distributed* λ-calculus [6] below refutes that argument. Let $a, b, c, \ldots, a', b', c', \ldots$ range over a set of nodes ($\mathfrak{N}$). Intuitively, t^a is the term t running on the node a. The $(\mu_\mathbf{d})$ rule below captures communication (or, $\underline{\text{m}}$obility, hence "μ") **without** side-effects.

Definition 1. *The distributed λ-terms are defined as: $\lambda_d \ni t^a = x^a \mid (\lambda x.t^a)^b \mid (t^a\ t^b)^c$. Reductions $. \xrightarrow{d} .$ on λ_d follow, where common capture-avoiding measure apply:*

$$\begin{array}{llllllll}
(\lambda x.t^a)^a & \xrightarrow{d}_\alpha & (\lambda y.t^a[y^a/x])^a & (\alpha_\mathbf{d}) & ((\lambda x.t_1^a)^a\ t_2^a)^a & \xrightarrow{d}_\beta & t_1^a[t_2^a/x] & (\beta_\mathbf{d}) \\
(\lambda x.(t^a\ x^a)^a)^a & \xrightarrow{d}_\eta & t^a & (\eta_\mathbf{d}) & t^a & \xrightarrow{d}_\mu & t^b & (\mu_\mathbf{d}).
\end{array}$$

Take λ_o for the set of ordinary λ-calculus terms and $\xrightarrow{o}$ for its reductions. Distributed λ-calculus is equivalent to the ordinary λ-calculus, and, is pure.

Theorem 2. *For every node a and b, the reduction $t \overset{o}{\twoheadrightarrow} t'$ implies $[\![t]\!]^{+a} \overset{d}{\twoheadrightarrow} [\![t']\!]^{+a}$, and, the reduction $t^a \overset{d}{\twoheadrightarrow} t'^b$ implies $[\![t^a]\!]^- \overset{o}{\twoheadrightarrow} [\![t'^b]\!]^-$.*

[*] This work is funded by the LightKone European H2020 project under grant agreement no. 732505.

The distributed λ-calculus is not general enough to model all distributed programming. In particular, it misses interaction with the outside world, e.g., human-beings. Such interactions are the **only** source of effectfulness. Other node communications can be programmed purely.

But, what if the effectfulness of communication can still go unobserved and be *view*ed as pure functional? That is certainly not possible universally, i.e., to every observer. Some observers might, nonetheless, not be concerned about the given communication and its side-effects.

Noticing that possible lack of concern drives this paper. We formalise what it means for a node not to be concerned about the side-effects of a given distributed program. Armed with that formalism, we present a new programming paradigm, in which the programmer is urged to mark the proportions of their code that are pure for a given node. The larger such proportions are, the more it is possible for the given node to benefit from the purity properties of the code. Typical benefits include the ones enumerated earlier for pure functional programming. Of course, it remains for the language to verify the correctness of purity markings.

Mind the subtle difference with monadic programming. Meritoriously, monads pretend there are no side-effects. They interpret an effectful computation as a state transition of a superficial universe. That is, monads are to deny the existence of side-effects (even though their type indicates the type of side-effects they have). In contrast, we acknowledge side-effects, yet, help the right nodes benefit from the purity of the correctly marked code proportions.

We are not the first to notice that effectfulness depends on the observer. Even though effectful, the Accelerate library [3] of Haskell has a purely functional interface, and, refrains from monadic treatments. That is becaus the only side effect of evaluating an Accelerate expression with 'run' is compilation and execution of the program. However, that side effect is not observable to Accelerate's host PL (i.e., Haskell).

In this paper, we present $\lambda(\mathsf{port})_\circ$: a variation of $\lambda(\mathsf{fut})$ [4] with built-in ports and pure blocks (i.e., blocks of code marked for purity). Ports are the only means for side-effects in $\lambda(\mathsf{port})_\circ$. In essence, the $\lambda(\mathsf{port})_\circ$ ports are asynchronous communication media. They are designed for interaction with our otherwise pure calculus. Sending to a port will add a message to the port's stream, which can be read by a pure expression (c.f. *srv* at Example 7).

One may wonder why we build on top of λ-Calculus as opposed to π-Calculus. Those two calculi were designed for modelling computations and agent systems, respectively. As such, exceptions [5, 1] aside, most PL results are established on top of the former. In this work we are particularly interested in the pure functional programming results. Our aim is to reuse those results. So, like more conventional λ-Calculi with futures [2, 4], we build on top of λ-Calculus.

We present the syntax and semantics of $\lambda(\mathsf{port})_\circ$ (Definitions 3 and 4). We formally define a notion of observational equivalence for a given node (Definition 5). Most importantly, we give an example where expanding a pure block goes unobserved by a node (Theorem 6). Finally, we showcase $\lambda(\mathsf{port})_\circ$ for a client-server application with only a couple of clients (Example 7).

2 Syntax and Semantics

The $\lambda(\mathsf{port})_\circ$ syntax is tailored for our minimal working example (Example 7), in particular.

Definition 3. *The $\lambda(\mathsf{port})_\circ$ expressions (E) and configurations (G) are defined below, where* this *font is for keywords:*

$$
\begin{array}{lll}
e & ::= & x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid f(x) = e \mid e_1; e_2 \mid e :: s \\
 & \mid & \mathsf{match}\ s\ \mathsf{for}\ \{x :: s' \Rightarrow e\} \mid \mathsf{send}\ \ \mathsf{to}\ p^b \mid \mathsf{pure}^{\overline{a}}\ \{e\}
\end{array}
\qquad
g \ ::= \ \mathsf{port}\ p^a \mid e \mid e^a \mid g_1 \parallel g_2
$$

Assume stream names $s, s', \ldots, s_1, s_2, \cdots \in S$, where E, S, and $\mathfrak{N}$ are disjoint. The syntax for an expression e is mostly routine: variables, constants, λ-abstractions, function applications, named functions ($f(x) = e$), sequential composition, and, *cons* expressions. Our pattern matching is less routine by only allowing a single match and only against cons expressions. Marking e's lack of side-effects for a list of nodes $\overline{a}$, if at all, is $\mathsf{pure}^{\overline{a}}\ \{e\}$. Write $\mathsf{pure}\ \{e\}$ to indicate universal purity of e. Purity markings are verified at runtime. Then, there are sending to ports, and pure blocks. Configurations are port declarations, (node-annotated) expressions, and concurrent compositions. One annotates an expression e with a node a as e^a. Such an annotation only applies to the outermost layer. Assuming port names $p, p', \ldots, p_1, p_2, \cdots \in P$, our ports $p^a, p'^a, \ldots, p^b, p'^b, \cdots \in P \times \mathfrak{N} = P\mathfrak{N}$ consist of their name and the node they belong to. We use structural congruence for concurrent compositions: $g_1 \parallel g_2$ and $g_2 \parallel g_1$ are the same. In $\lambda(\mathsf{port})_\circ$, the number of concurrent compositions is known statically.

We present a small-step operational semantics for the syntax given in § 2. Judgements $\textsc{j}, \textsc{j}', \cdots \in J$ of the operational semantics take the form $\tau : g \to \tau' : g'$. Each τ is a tuple (ρ, σ), where $\rho : S \rightharpoonup \{\bot\} \cup (E \times S)$ is an environment and $\sigma : P\mathfrak{N} \rightharpoonup S$ is a store. $\rho(s) = \bot$ denotes that s known to ρ but unbound in the current state of the program. The clause $\textsc{j} = \tau : g \to \tau' : g'$ sets $\textsc{j}$ as an alias for $\tau : g \to \tau' : g'$. Write $nc_j(\textsc{j})$ for the set of <u>n</u>odes (environment or store) bindings of which were <u>c</u>hanged over $\textsc{j}$. Write $g \to g'$ as a shorthand for $\tau : g \to \tau : g'$. Write $\to^*$ for the transitive and reflexive closure of $\to$. Finally, fix a set of values ranged over by $v_1, v_2, \ldots,$ including unit and all cs.

Definition 4. *Rules of the* $\lambda(\mathsf{port})_\circ$ *operational semantics follow.*

$$(\lambda x.e_1)\ e_2 \to e_1[e_2/x] \quad (\textsc{App-E}) \qquad f\ e \parallel f(x) = e' \to e'[e/x] \parallel f(x) = e' \quad (\textsc{App-F})$$

$$v; e \to e \quad (\textsc{Seq-1}) \qquad \mathsf{match}\ (e :: s)\ \mathsf{for}\ \{x :: s' \Rightarrow e'\} \to e'[e/x, s/s'] \quad (\textsc{Mat-1})$$

$$\mathsf{match}\ \bot\ \mathsf{for}\ \{x :: s \Rightarrow e\} \to \mathsf{match}\ \bot\ \mathsf{for}\ \{x :: s \Rightarrow e\} \quad (\textsc{Mat-2})$$

$$\frac{\tau : e_1 \to \tau' : e_1'}{\tau : e_1; e_2 \to \tau' : e_1'; e_2}\ (\textsc{Seq-2}) \qquad \frac{e \to e'}{e^a \to e'^a}\ (\textsc{Annot}) \qquad \frac{\tau : g_1 \to \tau' : g_1'}{\tau : g_1 \parallel g_2 \to \tau' : g_1' \parallel g_2}\ (\textsc{Cncr})$$

$$\frac{\textsc{j} = \tau : e \to \tau' : e' \quad \overline{a} \notin nc_j(\textsc{j})}{\tau : \mathsf{pure}^{\overline{a}}\ \{e\} \to \tau' : \mathsf{pure}^{\overline{a}}\ \{e'\}}\ (\textsc{Pure})$$

$$\frac{p, s\ \textit{fresh}}{(\rho, \sigma) : \mathsf{port}\ p^a \to (\rho[s \mapsto \bot], \sigma[p^a \mapsto s]) : \mathsf{unit}}\ (\textsc{Port})$$

$$\frac{\sigma(p^a) = s \quad \rho(s) = \bot \quad s'\ \textit{fresh}}{(\rho, \sigma) : \mathsf{send}\ e\ \mathsf{to}\ p^a \to (\rho[s' \mapsto \bot, s \mapsto e :: s'], \sigma[p^a \mapsto s']) : \mathsf{unit}}\ (\textsc{Send})$$

The rules in the first four lines are standard. Inside a block that is to be pure from the viewpoint of a, the rule (Pure) allows reductions that do not alter the parts of store that pertain to a. That is, they do not declare new ports for a; neither do they send to any of a's ports. According to (Send), when an expression e is sent to p^a, the respective stream s needs to be unbound. In such a case, we reduce by allocating a fresh and unbound stream s' and resetting the environment binding of s to $e :: s'$, hence, putting e at the front of the trailing stream.

3 Results

Write $\Delta^*_{\mathsf{port}}(\tau, g)$ for the nodes for which new ports were declared or the existing ports of which were sent to over $\tau : g \to^*_{\_} :\_$, where "_" is our wildcard notation. Intuitively, according to

Definition 5 below, a node observes two configurations equivalently when, for a single reduction step of one, the other can take one step (or more) with α-equivalent impacts on the parts of the environment and store that pertain to the node.

Definition 5. *Call g_1 and g_2 observationally equivalent **for a node** a (write $g_1 \sim_a g_2$) when*

$$\forall \tau. \begin{cases} \tau : g_1 \to \tau_1 : g_1' \Rightarrow \exists \tau_2, g_2.\ (\tau : g_2 \to^* \tau_2 : g_2') \wedge (\tau_1|_a \overset{\alpha}{\equiv} \tau_2|_a) \wedge (g_1' \sim_a g_2') \\ \tau : g_2 \to \tau_2 : g_2' \Rightarrow \exists \tau_1, g_1.\ (\tau : g_1 \to^* \tau_1 : g_1') \wedge (\tau_1|_a \overset{\alpha}{\equiv} \tau_2|_a) \wedge (g_1' \sim_a g_2') \end{cases} .$$

In words, the following theorem states that, when an expression is proceeded by a pure block, if the expression has no side-effect for the nodes the proceeding block is pure for, one can move the expression into the pure block; the result will be the same for those nodes.

Theorem 6. *Let $e_1, e_2 \in E$ and $\overline{a} \in \mathfrak{N}$. Then, $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{port}}(\tau, e_1)$ implies $e_1; \mathsf{pure}^{\overline{a}}\ \{e_2\} \sim_a \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$. Likewise, $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{port}}(\tau, e_2)$ implies $\mathsf{pure}^{\overline{a}}\ \{e_1\}; e_2 \sim_a \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$.*

Example 7. *Let $\mathfrak{N} = \{srv, c_1, c_2\}$, where srv is a server and c_1 and c_2 are clients. The internal states of the nodes are st_s, st_1, and st_2, respectively. Based on their own state, clients form a query and send it to the server's single port (p^{srv}). The server processes queries locally. Let f_c and f_s be **pure** client-side and server-side functions. Let also $\mathsf{stream}(p^a)$ denote the stream of p^a. Then, the client-server program below is pure everywhere, except upon: (1) declaring the port, and, (2) sending queries to the server – **exclusively** from the viewpoint of the server.*

$$\begin{aligned} \mathsf{port}\ p^{srv} &\parallel (srv\ st_s\ \mathsf{stream}(p^{srv})) \parallel (client\ st_1)^{c_1} \parallel (client\ st_2)^{c_2} \\ &\parallel client(st) = \mathsf{pure}^{\overline{c}}\ \{\mathsf{send}\ (query\ st)\ \mathsf{to}\ p^{srv};\ client\ (f_c\ st)\} \\ &\parallel srv\ (st, s) = \mathsf{pure}\ \{\mathsf{match}\ s\ \mathsf{for}\ \{q :: t \Rightarrow srv\ (f_s\ (q, st), t)\}\} \end{aligned}$$

□

References

[1] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *Proc. of 1st Int. Symp. on Agent Systems and Applications and 3rd Int. Symp. on Mobile Agents, ASA/MA '99*, pages 22–29. IEEE, 1999.

[2] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Program.*, 9(1):1–31, 1999.

[3] T. L. McDonell. *Optimising Purely Functional GPU Programs.* PhD thesis, Univ. of New South Wales, Sydney, 2015.

[4] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364(3):338–356, 2006.

[5] B. C. Pierce and D. N. Turner. Pict: A programming language based on the π-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[6] P. Van Roy. Why time is evil in distributed systems and what to do about it. CodeBEAM keynote talk, 2019. Available at https://www.youtube.com/watch?v=NBJ5SiwCNmU

Analyzing Usage of Data in Array Programs

Jan Haltermann

Dept. of Computer Science, Paderborn University, Germany
jfh@mail.upb.de

1 Introduction

Most static dataflow analyses, like constant propagation or live variable analysis, are designed for variables having primitive data types and are not applicable to arrays. For large arrays or in case that the length of an array is not known in advance, creating variables for each array index (array expansion) is infeasible or impossible. Treating the array as a single variable (array smashing) leads to imprecise results, when not all array elements are treated in the same way.

2 Approach

In this paper, we tackle the problem of analyzing programs containing arrays of unknown length and computing properties for their array elements by presenting two novelties: First, we provide a framework to analyze arrays by extending arbitrary program analyses for primitive variables to full arrays, based on the array segmentation domain presented by Cousot et al. [3]. Second, we develop the $\mathcal{C}_{\mathbb{LU}}$ analysis for identifying non-used array elements, as instance of the framework. To illustrate the framework and the analysis, we use the example given in Figure 1.

2.1 Framework for Analyzing Arrays

To infer properties for array elements in programs, we make use of the array segmentation domain defined by Cousot et al. [3]. Intuitively, an array segmentation splits an array into several segments, such that each element from the array is associated to exactly one segment. The information computed by an analysis is also associated with one segment and hence the full array is covered The array segmentation domain D is defined as follows:

$$D = [E : (\underbrace{\{e_1^0, \dots\}}_{B_0}\ p_0[?]_0\ B_1\ p_1[?]_1\ \ \dots\ \ p_{n-1}[?]_{n-1}\ \underbrace{\{e_1^n, \dots\}}_{B_n} \cup \square) \cup (\bot, \top)]^n$$

The bottom element is used to indicate unreachable path, the top element to avoid under-approximation analysis results. A segment describes an interval in the array, having a lower bound B_i of the segmentation being included and the upper bound B_{i+1} that is not included. Moreover, it contains some analysis information p_i from the underlying domain forming a complete lattice and a flag $'?'_i$, indicating if the segment may be empty. The segment bounds are simple binary expressions over integers, are ordered ascending and all expressions present in the same segment bound are assumed to haven an equal value. For example, the segmentation $\{0\}\ p_0?\ \{i\}\ p_1\ \{a.len()\}$ may be computed during the analysis of the example program shown in Figure 1. It states that the information p_0 holds for the interval $[0, i[$, that might be empty and p_1 for all elements in the interval $[i, a.len()[$, containing at least one element.

The transfer relation used to compute successor segmentation when analyzing a program mainly retains the relation between the expressions present in the segment bounds. When merging or comparing two segmentation, we apply a unification algorithm. It ensure that both

```
int sum(int[] a)
    int res = 0;
    for( int i = 0; i < a.len()−1 ; i++)
        res = res + a[i];
    return res;
```

Figure 1: A program, computing the sum of all array elements except the last one

segmentation contain identical segment bounds, to be able to apply the merge or comparison segment-wise, making use of the underlying domain. The unification algorithm tries to maintain as many segment bounds as possible while guaranteeing an unique ordering of the segment bounds in both segmentation. Moreover, it guarantees that the number of segment bounds in a segmentation is bounded above. Formal definitions for enhanced versions of both, transfer function and unification algorithm can be found in the Masters Thesis of Haltermann, where the basic ideas are taken from Cousot [4].

To compute more precises results, we extended Cousot et al.'s domain in several ways: First, we add the explicit symbol $\Box$ to indicate, that the array can be empty. Thereby, more precise results are computed by removing doubtful ′?′'s. Second, we introduce the split-condition E together with using multiple segmentation for one program location. A split condition states in general an assumption on the array length, e. g. if it is less, equal or greater than a constant. By analyzing a segmentation w. r. t. the split condition, an analysis computes multiple different array segmentation for a single program location. Applying splitting allows us to improve the results, i. e. when analyzing programs containing branches having constant values in their conditions. Third, we added a strengthening mechanism using the path formula introduced by Beyer et al. [2] to enhance the performance of the analysis. Thereby, we can again remove doubtful ′?′'s, reducing the over-approximation of the analysis for the computed results.

2.2 $\mathcal{C}_{\mathbb{LU}}$ Analysis: Instantiation to Analyze Array Content Usage

The combined location and usage analysis $\mathcal{C}_{\mathbb{LU}}$ is an instantiation of our framework detecting unused array elements. An array index at position i is considered as used, if there are two program executions started with same values for all input elements except the value at index i and returning different values. The underlying domain detecting usage consists of the two elements **U**sed and **N**ot-used, where $\bot = N \sqsubseteq U = \top$. The transfer function and merge operator designed for single elements are given in [4]. The analysis is designed as a may analysis, hence the set of used array elements is over-approximated. Initially, every $\mathcal{C}_{\mathbb{LU}}$ analysis is started with the empty segmentation $\{0\}N?\{a.len()\}$, meaning that all elements are not used or the array is empty. When applied to the running example from Figure 1, we obtain the segmentation $\{\{0\}\ U?\ \{a.len() - 1,\ i\}\ N\ \{a.len()\},\ \Box\}$. Hence, we can state that all array elements in the potentially empty interval $[0, a.len() - 1[$ are used and one element in the interval $[a.len() - 1, a.len()[$ is not considered during computation, if the array is not empty.

In addition, we prove that any instantiation of the framework terminates, when providing a monotone transfer function for the underlying domain. We show that the analysis can only produce a finite number of segmentation, even for programs containing loops. During computation, a segmentation's length is bounded above by the number of statements on the longest, loop-free path leading to a location plus a constant value. This property of the framework is guaranteed by the unification procedure. Moreover, we show that the $\mathcal{C}_{\mathbb{LU}}$-analysis is correct in the sense that all results computed in fact over-approximates the set of used variables.

3 Implementation

We implemented the framework as a configurable analysis and the $\mathcal{C}_{\mathbb{LU}}$ analysis as instantiation in the CPAchecker[1] [1]. First evaluations on small, hand crafted examples containing loops and branches like in Figure 1 and patterns extracted by Xiao et al. [5] during their study of reducers, has lead to precise results. For instance, when applied to the running example, the $\mathcal{C}_{\mathbb{LU}}$ analysis computed a precise result in around one second.

4 Conclusion and Further Work

We presented a lightweight framework usable to easily lift analyses developed for single variables to arrays, additionally guaranteeing termination. The framework is based on the array segmentation domain introduced by Cousot et al., heavily extended to obtain more precise results. Moreover, we presented an instantiation of the framework to detect usage of array elements and demonstrated it on a small example. Both, the framework as well as the $\mathcal{C}_{\mathbb{LU}}$ analysis are implemented and evaluated using the CPAchecker.

In the future, we want to enhance the $\mathcal{C}_{\mathbb{LU}}$ analysis to be able to detect non-commutative usage of array elements. Moreover, we plan to investigate if using further analyses like interval analysis or constant value analysis, can either lead to more precise results or speed up the computation time.

References

[1] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Proc. of 19th Int. Conf. on Computer-Aided Verification, CAV 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[2] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. of 10th Int. Conf. on Formal Methods in Computer-Aided Design, FMCAD 2010*, pages 189–197. IEEE, 2010.

[3] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '11*, pages 105–118. ACM, 2011.

[4] Jan Haltermann. Analyzing data usage in array programs. Master's thesis, Paderborn University, 2019.

[5] Tian Xiao, Jiaxing Zhang, Hucheng Zhou, Zhenyu Guo, Sean McDirmid, Wei Lin, Wenguang Chen, and Lidong Zhou. Nondeterminism in MapReduce considered harmful? An empirical study on non-commutative aggregators in MapReduce programs. In *Proc. of 36th Int. Conf. on Software Engineering, ICSE '14*, pages 44–53. ACM, 2014.

[1] https://cpachecker.sosy-lab.org/

Syntactic Theory Functors for Specifications with Partiality

Magne Haveraaen[1], Markus Roggenbach[2], and Håkon Robbestad Gylterud[1]

[1] Dept. of Informatics, University of Bergen, Norway
[2] Dept. of Computer Science, Swansea University, United Kingdom
magne.haveraaen@uib.no, m.roggenbach@swan.ac.uk, hakon.gylterud@uib.no

Abstract

Here we suggest using “syntactic theory functors” (STFs) as a means to organise algebraic specifications of partial data types. Algebraic specifications of partial data types often introduce much syntactic clutter and is error prone in itself. STFs are a syntactic structuring mechanism for specifications, allowing systematic treatment of such specifications to avoid the pitfalls. As a syntactic structuring mechanism the STF can be left in the specification text to indicate how the partial specification will be encoded. The STF can also be expanded to expose the full specification text with all the detail and clutter in place.

1 Motivation

Algebraic specifications are a useful formalism for defining APIs and generic code. Such specification formalisms include the very simple equational and conditional equational logics, the logic of boolean expressions (quantifier free first order predicate logic) familiar to software developers, and standard first order predicate logic which is preferred by pre/post specifications. Thus algebraic specifications subsume pre/post specifications, which are focussed on the individual algorithm and not the API as such.

The weakness of algebraic specifications is in handling partiality or preconditions. While algebraic specifications are good at defining properties like what is the first element of a queue, or that two operations are inverses, like popping versus pushing an element on a stack, or division versus multiplication, they are not good for defining special cases like the first element of an empty queue, popping an empty stack, or dividing by zero. The former can be handled by choosing some arbitrary designated value as the first element of an empty queue, popping an empty stack can be kludged to be an empty stack, but division by zero has no viable answer since zero is an absorbing element for multiplication and thus causes deeper problems.

There are many approaches to partiality in algebraic specifications, see [4] for a discussion. These approaches turn out to introduce “syntactic clutter” in the machinery used for dealing with partiality, making it difficult to convey the important properties of an API, and can be very error prone in getting right. Here is a straight forward stack specification.

```
specification Stack = type S, E;
  function new: -> S;        function push: S, E -> S
  function pop: S -> S;      function top: S -> E
  axiom s:S, e:E pop(push(s,e)) = s;       axiom s:S, e:E top(push(s,e)) = e
```

The error algebra version deals with the problematic cases of the empty stack.

```
specification Stack_error = type S, E
  function new: -> S;        function push: S, E -> S
  function pop: S -> S;      function top: S -> E
  % New error constants
```

```
function error_s : -> S;   function error_e : -> E
% New axioms defining the error situations
axiom pop(new()) = error_s();           axiom top(new()) = error_e()
% Error propagation axioms for each of the operations of the Stack signature
axiom s:S push(s,error_e()) = error_s(); axiom e:E push(error_s(),e) = error_s()
axiom pop(error_s()) = error_s();       axiom top(error_s()) = error_e()
% Protected axioms from the Stack specification
axiom s:S, e:E s!=error_s() && e!=error_e() => pop(push(s,e)) = s
axiom s:S, e:E s!=error_s() && e!=error_e() => top(push(s,e)) = e
```

Here we can deduce that `pop(push(s,error_e()))=error_s()`, but cannot deduce that `pop(push(s,error_e()))=s`. Without this care in introducing the appropriate prerequisits, the latter deduction would imply that `s=error_s()` for all stacks `s`.

We suggest applying "syntactic theory functors" (STFs) as a means to organise such specifications. STFs are a syntactic structuring mechanism for specifications and were introduced at NWPT'18. They allow a systematic manipulation of declarations, adding new and modifying existing axioms of a specification. As syntactic structuring mechanisms they can be kept in the specification text thus indicating which partiality mechanism will be used, or they can be expanded giving the correspondingly "cluttered" specification in the semantically intended way. Careful design of the STFs will ensure the intended specification, avoiding inadvertent mistakes.

2 Syntactic Theory Functors

Syntactic theory functors are an institution [1] independent syntactic structuring mechanism. STFs work on discrete institutions, i.e., any logic with a model theory, e.g., a programming system, consisting of

- Interface declarations, called *signatures* $\mathbf{Sig}_{\mathrm{id}}$ or APIs.
- A specification formalism for(Σ) which defines the set of *formulae* for each signature Σ.
- A collection of *models* mod(Σ) for each signature.
- A *satisfaction relation* $\models_\Sigma \subseteq \mathrm{mod}(\Sigma) \times \mathrm{for}(\Sigma)$ that defines which models satisfy a formulae.

A *theory* $\langle \Sigma, \Phi \rangle$ consists of a signature Σ and a set of formulae $\Phi \subseteq \mathrm{for}(\Sigma)$. A specification structuring mechanism is syntactic if, when applied to a theory, it can by expanded (flattened) to a theory, i.e., it can be seen as a mapping from theories to theories. The STFs are theory mappings with additional requirements.

Definition 1 (Syntactic Theory Functor (STF))**.** *Let* $\mathbf{Th}_{\mathrm{id}}$ *denote the theories of an institution. A mapping* $F : \mathbf{Th}_{\mathrm{id}} \to \mathbf{Th}_{\mathrm{id}}$ *from theories to theories is a* syntactic theory functor *if the application of* F *on theories can be decomposed*

$$F(\Sigma, \Phi) = \langle F_{\mathtt{sig}}(\Sigma), F_{\mathtt{base}}(\Sigma) \cup F_{\mathtt{for},\Sigma}(\Phi) \rangle$$

where

- $F_{\mathtt{sig}} : \mathbf{Sig}_{\mathrm{id}} \to \mathbf{Sig}_{\mathrm{id}}$ *is a mapping of signatures to signatures,*
- $F_{\mathtt{base}} : \mathbf{Sig}_{\mathrm{id}} \to \mathbf{Set}$ *is a mapping from signatures to sets of formulae with* $F_{\mathtt{base}}(\Sigma) \subseteq \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$*, and*

- $F_{\mathtt{for},\Sigma} : \mathrm{for}(\Sigma) \to \mathrm{for}(F_{\mathtt{sig}}(\Sigma))$ *is a function for every signature* Σ*.*

STFs are additive, i.e., $F(\Sigma, \Phi_1 \cup \Phi_2) = F(\Sigma, \Phi_1) \cup F(\Sigma, \Phi_2)$. STFs compose, i.e., given STFs F and G, then $F; G$ is also an STF. Some STFs are model consistent, i.e., whenever $F(\Sigma, \Phi) = \langle \Sigma', \Phi' \rangle$, then there is a mapping $\mu : \mathrm{mod}(\Sigma') \to \mathrm{mod}(\Sigma)$ s.t. $\mu(M') \models_\Sigma \varphi \iff M' \models F(\Sigma, \{\varphi\})_2$. Model consistent STFs build institutions with non-trivial morphisms between signatures and corresponding morphisms on formulae and models. Such institutions have a very well behaved translation of formulae and models between different signatures. STFs that do not build institutions may have a weaker reuse of formulae, e.g., they may preserve or reflect only some kind of formulae from one signature to another. The STFs for partial specifications will normally not be model consistent, as the purpose is to allow models which are less constrained for the arguments that break preconditions than for those arguments that satisfy the precondition—a flexibility not allowed by the source specification.

3 Approaches to partiality and corresponding STFs

In the paper [4] Peter Mosses presents succinctly a list of approaches to partial specifications: error algebras, ok-predicates, exception algebras, labeled algebras, various forms of order sorted algebras, and classified algebras. We will develop STFs for most of these formalisms, showing that the notational complexity and error proneness of writing these specifications can be handled nicely. Finally we will look into guarded algebras [2].

To illustrate the approach we will sketch an error STF. The error STF is parameterised by a listing earg of the error constants and the axioms that returns the error element. It does the following transformation to a specificiation.

- $\mathrm{error}_{\mathrm{earg,sig}}$ create a new signature where the provided error constants have been added to the signature.
- $\mathrm{error}_{\mathrm{earg,base}}$ add the new axioms and error propagation rules for each of the existing operations.
- $\mathrm{error}_{\mathrm{earg,for}}$ modify each existing axiom by prepending checks for the relevant error elements.

If the given parameter list does not appropriately match the argument theory, the error STF will provide relevant error messages. For the stack example the parameter list might be:

```
earg = type S, E; function error_s : -> S; function error_e : -> E % The error elements
  % Axioms defining the error situations
  function new: -> S;           function pop: S -> S;           function top: S -> E
  axiom pop(new()) = error_s(); axiom top(new()) = error_e()
```

Expanding the error STF on the `Stack` specification yields the `Stack_error` specification above.

4 Tools for reasoning

The translations for handling partiality in algebraic specifications typically involve enlarging the API (signatures) of the specification, adding some axioms based on the declarations, and modifying axioms by introducing conditionals.

Now many of the standard reasoning tools have problems dealing with large sets of conditional formulae. Here we discuss some of these difficulties and point to tools and methods which alleviate these problems.

5 Summary

In this presentation we will demonstrate how STFs can simplify writing partial specifications, making them more readable and comprehensible by not cluttering up the specifications. We show this for a range of approaches to partiality in algebraic specifications.

We also give examples of tool support for reasoning about such partial specifications, and prove/disprove simple claims about the transformed specifications at the STF level.

References

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

[2] M. Haveraaen and E.G. Wagner. Guarded algebras: Disguising partiality so you won't know whether it's there. In D. Bert, C. Choppy, P. Mosses, editors, *Selected Papers from 14th Int. Workshop on Algebraic Development Techniques, WADT '99*, volume 1827 of *Lecture Notes in Computer Science*, pages 182–200. Springer, 2000.

[3] P. Mosses. *CASL Reference Manual: The Complete Documentation of the Common Algebraic Specification Language*, volume 2960 of *Lecture Notes in Computer Science.* Springer, 2004.

[4] P. Mosses. The use of sorts in algebraic specifications. In M. Bidoit and C. Choppy, editors, *Selected Papers from 8th Workshop on Specification of Abstract Data Types and 3rd COMPASS Workshop, ADT/COMPASS '91*, volume 655 of *Lectures Notes in Computer Science*, pages 66–92. Springer, 1993.

Operational Semantics with Semicommutations

Hendrik Maarand[1] and Tarmo Uustalu[2,1]

[1] Department of Software Science, Tallinn University of Technology, Estonia
hendrik@cs.ioc.ee
[2] Department of Computer Science, Reykjavik University, Iceland
tarmo@ru.is

1 Introduction

Consider the following program representing the message-passing pattern.

$$x := 41; y := 1 \parallel r1 := y; r2 := x$$

Here the first thread stores to x the result of some computation and then sets y to 1 to indicate that it has finished the computation. Intuitively, starting from an initial state where everything is set to zero, this program should not reach a final state where $r1 = 1$ and $r2 = 0$. If we apply an optimization (either in advance or at runtime) which happens to reorder the two instructions in either of the threads, then the final state in question becomes valid. For example, the optimized program could be the following where the instructions of the second thread have been reordered.

$$x := 41; y := 1 \parallel r2 := x; r1 := y$$

While the programs $r1 := y; r2 := x$ and $r2 := x; r1 := y$ give the same result in a sequential setting, executing them in parallel with $x := 41; y := 1$ might give different results. Namely, the optimized program allows the final state where $r1 = 1$ and $r2 = 0$.

The phenomenon described above where "safe" optimizations applied to individual threads of a concurrent program leads to additional behaviours is often referred to as relaxed memory models (relaxed in the sense that more behaviours are allowed).

In the following we describe an operational semantics which is able to capture some optimizations like the one described above. The main idea is that given a program $p; q$ we can, under certain conditions, execute an instruction from q even when p is not yet fully executed. This is an extension of our earlier work [2] where we defined reordering derivative-like operations on regular expressions.

2 Preliminaries

A semicommutation alphabet is an alphabet Σ together with an irreflexive relation $\theta \subseteq \Sigma \times \Sigma$ which is called the semicommutation relation [1]. We write $\theta(a, b)$ for $(a, b) \in \theta$. We extend θ to words and letters by $\theta(\varepsilon, a) =_{\mathrm{df}} \mathsf{tt}$ and $\theta(ub, a) =_{\mathrm{df}} \theta(u, a) \wedge \theta(b, a)$.

The set RES of *regular expressions with shuffle* over an alphabet Σ is given by the grammar:

$$E ::= a \in \Sigma \mid 0 \mid E + E \mid 1 \mid EE \mid E^* \mid E \parallel E.$$

In the following we assume a set Σ of instructions (ranged over by $a, b, c, \ldots$) and a set $\mathbb{S}$ of machine states (ranged over by σ). The set of booleans is denoted by $\mathbb{B}$. The interpretation of instructions as (partial) state transformers is given by $[\![\_]\!] : \Sigma \to \mathbb{S} \rightharpoonup \mathbb{S}$. We use the notation $(\sigma)a =_{\mathrm{df}} [\![a]\!]\sigma$ and extend it to words as $(\sigma)\varepsilon =_{\mathrm{df}} \sigma$ and $((\sigma)u)v =_{\mathrm{df}} (\sigma')v$ if $(\sigma)u = \sigma'$ and $\perp$ otherwise. We write $(\sigma)u{\downarrow}$ to express that $(\sigma)u$ is defined. We consider expressions $E \in \mathsf{RES}$ to be (concurrent) programs over the alphabet of instructions Σ.

3 Semantics

We consider an execution of a program to be a sequence of instructions applied to the initial state. Our goal is to describe the execution of programs in the "relaxed" manner described in the introduction. The question now is: given a program, how to determine those instructions that can be executed next? Intuitively, we allow to execute an instruction a before executing the preceding instructions u when we know that both ua and au lead to the same state.

Definition 1. *A semicommutation relation θ is* conservative *when, for every $a, b \in \Sigma$, we have $\theta(a, b) \Longrightarrow \forall\sigma.(\sigma)ab = (\sigma)ba$.*

It follows that if θ is conservative, then $\theta(u, a)$ implies $(\sigma)ua = (\sigma)au$. Thus we take $\theta(u, a)$ to be the justification that allows us to reorder a before u in the sequence of instructions ua. We now extend this idea to programs: given a program Ea we would like to find a program E' such that every execution u of E' is also an execution of E and a is reorderable with u.

Definition 2. *The* nullability *(empty word property) of a program and the θ-*reorderable part *of a program are given by the functions $\varepsilon : \mathsf{RES} \to \mathbb{B}$ and $R^\theta : \mathsf{RES} \times \Sigma \to \mathsf{RES}$ defined recursively by*

$$\begin{array}{rcl@{\qquad}rcl}
\varepsilon(b) & =_{\mathrm{df}} & \mathsf{ff} & R^\theta_a b & =_{\mathrm{df}} & \mathsf{if}\ \theta(b,a)\ \mathsf{then}\ b\ \mathsf{else}\ 0 \\
\varepsilon(0) & =_{\mathrm{df}} & \mathsf{ff} & R^\theta_a 0 & =_{\mathrm{df}} & 0 \\
\varepsilon(E+F) & =_{\mathrm{df}} & \varepsilon(E) \vee \varepsilon(F) & R^\theta_a(E+F) & =_{\mathrm{df}} & R^\theta_a E + R^\theta_a F \\
\varepsilon(1) & =_{\mathrm{df}} & \mathsf{tt} & R^\theta_a 1 & =_{\mathrm{df}} & 1 \\
\varepsilon(EF) & =_{\mathrm{df}} & \varepsilon(E) \wedge \varepsilon(F) & R^\theta_a(EF) & =_{\mathrm{df}} & (R^\theta_a E)(R^\theta_a F) \\
\varepsilon(E^*) & =_{\mathrm{df}} & \mathsf{tt} & R^\theta_a(E^*) & =_{\mathrm{df}} & (R^\theta_a E)^* \\
\varepsilon(E \parallel F) & =_{\mathrm{df}} & \varepsilon(E) \wedge \varepsilon(F) & R^\theta_a(E \parallel F) & =_{\mathrm{df}} & R^\theta_a E \parallel R^\theta_a F
\end{array}$$

Note that $R^\theta_a E$ just replaces those letters b in E with 0 for which $\theta(b, a)$ does not hold. Nullability is essentially a special case of θ-reorderability where we require $\theta(b, a)$ for every $a \in \Sigma$ (i.e., all letters in the expression would be replaced with 0).

Definition 3. *The θ-*reordering operational semantics *of a program is given by the relation $\to^\theta \subseteq \mathbb{S} \times \mathsf{RES} \times \Sigma \times \mathbb{S} \times \mathsf{RES}$:*

$$\frac{(\sigma)a{\downarrow}}{\langle \sigma, a\rangle \to^\theta (a, \langle (\sigma)a, 1\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E+F\rangle \to^\theta (a, \langle \sigma', E'\rangle)} \quad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, E+F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, EF\rangle \to^\theta (a, \langle \sigma', E'F\rangle)} \quad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, EF\rangle \to^\theta (a, \langle \sigma', (R^\theta_a E)F'\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E^*\rangle \to^\theta (a, \langle \sigma', (R^\theta_a E)^* E' E^*\rangle)}$$

$$\frac{\langle \sigma, E\rangle \to^\theta (a, \langle \sigma', E'\rangle)}{\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle \sigma', E' \parallel F\rangle)} \quad \frac{\langle \sigma, F\rangle \to^\theta (a, \langle \sigma', F'\rangle)}{\langle \sigma, E \parallel F\rangle \to^\theta (a, \langle \sigma', E \parallel F'\rangle)}$$

A configuration $\langle \sigma, E\rangle$ is terminal when $\varepsilon(E)$. If θ is the empty relation, then we get the ordinary operational semantics where nothing is reordered, i.e., $R^\theta_a E$ essentially becomes the condition $\varepsilon(E)$. If a derivation sequence starts with $\langle \sigma, E \rangle$ and ends with a terminal configuration $\langle \sigma', E' \rangle$, then σ' is a final state of the program E executed from the initial state σ.

Consider as an example the program $ab \parallel (c+d)ef+g$ and say that the next instruction we wish to execute is e (which is not the first instruction of the second thread). If, for example, we have $\theta(c,e)$, $(\sigma)e{\downarrow}$ and $\neg\theta(d,e)$, then it is the case that

$$\langle \sigma, ab \parallel (c+d)ef+g \rangle \to^{\theta} (e, \langle (\sigma)e, ab \parallel (c+0)1f \rangle).$$

Note that $R^{\theta}_{e}(c+d) = (c+0)$ since we have $\theta(c,e)$ but not $\theta(d,e)$. It can be shown that $ab \parallel (c+0)1f$ is equivalent to $ab \parallel cf$. The instruction g disappeared from the expression since the rules for $+$ resolve the nondeterminism and the instruction d disappeared from the expression since it was a predecessor of e and reordering e with d was not justified.

We now return to the example program from the introduction. Note that here we use ; to denote multiplication. Assume θ such that $\theta(r1 := y, r2 := x)$ (this θ could be the *concurrent-read-exclusive-write* condition) and let σ be the initial state where all variables are set to 0. This allows the following derivation sequence (we have omitted the labels on $\to^{\theta}$).

$$\begin{array}{lll}
\langle \sigma, & x := 41; y := 1 \parallel r1 := y; r2 := x \rangle & \to^{\theta} \\
\langle \sigma[r2 \mapsto 0], & x := 41; y := 1 \parallel r1 := y; 1 \rangle & \to^{\theta} \\
\langle \sigma[r2 \mapsto 0][x \mapsto 41], & 1; y := 1 \parallel r1 := y; 1 \rangle & \to^{\theta} \\
\langle \sigma[r2 \mapsto 0][x \mapsto 41][y \mapsto 1], & 1; 1 \parallel r1 := y; 1 \rangle & \to^{\theta} \\
\langle \sigma[r2 \mapsto 0][x \mapsto 41][y \mapsto 1][r1 \mapsto 1], & 1; 1 \parallel 1; 1 \rangle &
\end{array}$$

Since $\varepsilon(1;1 \parallel 1;1)$, we have that the program $x := 41; y := 1 \parallel r1 := y; r2 := x$ executed from the initial state σ leads to a final state where $r1 = 1$ and $r2 = 0$.

4 Future Work

Our plan is to extend this framework so that it is possible to describe (a large part of) the multicopy-atomic ARMv8 memory model [3] and then check this description against the existing litmus-tests for this memory model.

One possible extension is to include reordering actions in the framework. For example, when $\theta(a,b)$ and we reorder b before a, then the instruction a acts on b from the left and b acts on a from the right. Now $x := 1; y := x$ can be reordered as $y := 1; x := 1$. The instruction $y := x$ becomes $y := 1$ and thus reads the value written by $x := 1$ before it has reached the memory.

Another extension we are considering is using a state-dependent θ, i.e., for every state σ we may have a different semicommutation relation θ_{σ}. This is useful for describing allowed reorderings for instructions like $x := [y]$ which stores to variable x the value read from the memory location whose address is given by the variable y.

Acknowledgments This research was supported by the ERDF funded Estonian national CoE project EXCITE (2014-2020.4.01.15-0018) and the the Estonian Ministry of Education and Research institutional grant no. IUT33-13.

References

[1] Mireille Clerbout and Michel Latteux. Semi-commutations. *Inf. Comput.*, 73(1):59–74, 1987.

[2] Hendrik Maarand and Tarmo Uustalu. Reordering derivatives of trace closures of regular languages. In Wan Fokkink and Rob van Glabbeek, editors, *Proc. of 30th Int. Conf. on Concurrency Theory, CONCUR 2019*, volume 140 of *Leibniz International Proceedings in Informatics*, pages 40:1–40:16. Dagstuhl Publishing, 2019.

[3] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018.

Composition of Multilevel Modelling Hierarchies

Alejandro Rodríguez[1], Adrian Rutle[1], Francisco Durán[2],
Lars Michael Kristensen[1], Fernando Macías[3], and Uwe Wolter[4]

[1] Western Norway University of Applied Sciences, Bergen, Norway
[2] Universidad de Málaga, Spain
[3] Universidad de Extremadura, Cáceres, Spain
[4] Dept. of Informatics, University of Bergen, Norway

Introduction. Model-driven software engineering (MDSE) has been proven to be a successful approach in terms of a gain in quality and effectiveness [15]. It tackles the constantly growing complexity of software by utilizing abstractions and modelling techniques.

MDSE promotes separation of concerns to better handle the complexity of software systems, which in turn leads to the creation of several models that need to be composed when reasoning about the overall system. Traditional MDSE approaches such as the Eclipse Modelling Framework (EMF) [14] and the Unified Modelling Language (UML) [1] are based on the two-level modelling approach: one for (meta)models and one for their instances. This enforces model designers to specify systems within only two abstraction levels, which in several situations may raise challenges like model convolution and accidental complexity [5]. Multilevel modelling (MLM) addresses these challenges by eliminating the restriction in the number of times a model element can be instantiated. Indeed, MLM has proven to be a successful approach in areas such as software architecture and process modelling domains [5, 2]. In this context, MLM techniques match well with the creation of domain-specific modelling languages (DSMLs), especially when we focus on behavioural languages since behaviour is usually defined at the metamodel level while it is executed at least two levels below; i.e., at the instance level.

Our approach for MLM. It is based on the idea that one must be able to specify models which are both generic and precise [8]. This encompasses not only the definition of structure but also behaviour. We specify behaviour descriptions by defining in-place model transformations (MTs) which are rule-based modifications of an initial model that give rise to a transition system. We have proposed in previous work the so-called Multilevel Coupled Model Transformations (MCMTs) as a means to overcome the issues of both the traditional two-level transformation rules and the multilevel model transformations. While the former lacks the ability to capture generalities, the later is too loose to be precise enough (case distinctions) [9]. We use our own tool MultEcore [7] to specify both the structure and the semantics of multilevel hierarchies and rely on our infrastructure which utilizes Maude as an execution engine [12]. The Maude specification automatically created by MultEcore can be used for simulation and analysis [11]. It is also possible to further conduct reachability analysis and model checking. While the former can be done by means of strategies [4], the latter can be performed through the model checker that Maude implements.

Composition of MLM DSMLs. One of the most successful techniques in MDSE is the definition of DSMLs. Even though they aimed to specific domains, many of them share certain commonalities coming from similar modelling patterns [10]. Needless to say, composition is key in achieving interoperability among these DSMLs. In this paper, we focus on the theoretical constructions for composition of multilevel DSMLs by presenting two approaches (depicted in Fig 1). Our framework is founded on graph transformations and category theory. The composition of modelling hierarchies would be carried out by pushout construction in the category of graph chains (see [6]), while the composition of the transformation rules (MCMTs) would be carried out by the amalgamation of these rules. We plan to formalise the latter by extending our formalisation through an adaptation of the constructions in [3] where the amalgamation of two-level DSMLs is formally described.

Several approaches pursue composition of languages by defining a merge operator. Intuitively, “the common elements are included only once, and the other ones are preserved”. Fig. 1a depicts how merge would be defined over two multilevel hierarchies, each one representing a DSML. The Merged Hierarchy is the result of merging the involved multilevel hierarchies. Instance models can now instantiate elements (dashed arrows represent typing graph homomorphisms) that come from the merging process. These instances can be executed producing the Transition system which is obtained by applying the rules that come from the amalgamation of the rule sets (RS) of each hierarchy ($RS_1 \uplus RS_2$).

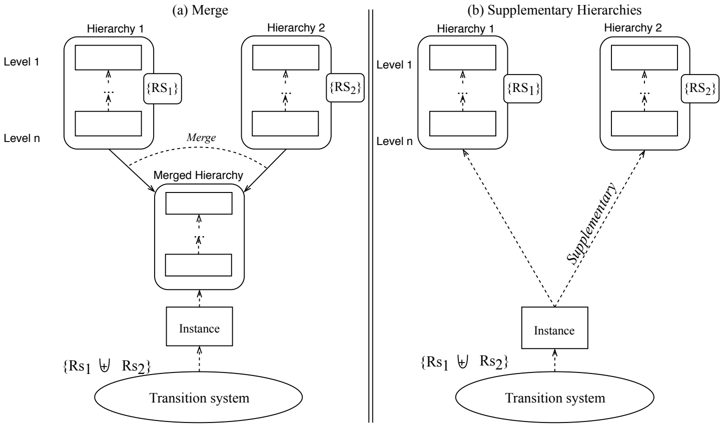

Figure 1: Conventional merge vs supplementary referencing composition

Our approach for composition. In our approach, the modeller typically works with a multilevel hierarchy which we identify as the *application hierarchy*. Application hierarchies can optionally include an arbitrary number of *supplementary hierarchies* which add new dimensions to the application one. In [13] we show how several supplementary hierarchies are applied to domain-specific Coloured Petri Nets. This allows model elements to have at least one type from the levels above in the application hierarchy and potentially one other type per incorporated supplementary hierarchy. Although the use of supplementary hierarchies was a design choice to facilitate the addition of supplementary features to a functional main language, we are now investigating how to extend and formalize their usage for the composition of structure and behaviour of MLM hierarchies. We consider our approach as a realization of the composition process by taking advantage of the supplementary hierarchies and double typing. This is shown in Fig. 1b, where we aim to build the composition by assigning more than one type to elements in the Instance level. In this case, Hierarchy 2 is considered supplementary and its elements can be used to add additional types to elements in the Instance model.

When it comes to structure composition, we can compare the use of supplementary hierarchies to the *Aggregation* scenario depicted in [10], where a language uses some constructs provided by other languages. With our approach, the additional languages (provided by the supplementary types) can be added/removed consistently which provides a strong separation of concerns and strengthen reusability. Hence, we use a “virtual” merge in which we achieve composition by relying on type combinations. Our goal now is to further investigate and decide which of the approaches depicted in Fig. 1 suits best.

When it comes to behaviour (MCMTs) composition, we analyse which two types (from each hierarchy) are used to double-type an element, and use this information to guide the amalgamation of the rules at runtime. The amalgamation process is based on double-typing which in turn is equivalent to type-sameness (commonality model for constructing the pushout) on which traditional merging is found.

Our next steps towards the formalization of the composition of multilevel DSMLs is twofold: (1) to determine which of the techniques shown in Fig. 1 is more convenient; and (2) to define which rules can be amalgamated, identify limitations and corner cases of the approach and determine coordination mechanism for the application of the rules.

References

[1] UML. http://www.uml.org/

[2] C. Atkinson and T. Kühne. On evaluating multi-level modeling. In L. Burgueño et al., eds., *Proc. of MODELS 2017 Satellite Events*, v. 2019 of *CEUR Wksh. Proc.*, pp. 274–277. RWTH Aachen, 2017.

[3] F. Durán, A. Moreno-Delgado, F. Orejas, and S. Zschaler. Amalgamation of domain specific languages with behaviour. *J. Log. Algebr. Methods Program.*, 86(1):208–235, 2017.

[4] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. *Electron. Notes in Theor. Comput. Sci.*, 174(11):3–25, 2007.

[5] J. d. Lara, E. Guerra, and J. S. Cuadrado. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2), article 12, 2014.

[6] F. Macías. *Multilevel modelling and domain-specific languages.* PhD thesis, Western Norway University of Applied Sciences and University of Oslo, 2019.

[7] F. Macías, A. Rutle, and V. Stolz. Mult4core: Combining the best of fixed-level and multilevel metamodelling. In C. Atkinson et al., eds., *Proc. of 3rd Int. Wksh. on Multi-Level Modelling, MULTI 2016*, v. 1722 of *CEUR Wksh. Proc.*, pp. 66–75. RWTH Aachen, 2016.

[8] F. Macías, A. Rutle, V. Stolz, R. Rodriguez-Echeverria, and U. Wolter. An approach to flexible multilevel modelling. *Enterp. Modelling Inf. Syst. Archit.*, 13, article 10, 2018.

[9] F. Macías, U. Wolter, A. Rutle, F. Durán, and R. Rodriguez-Echeverria. Multilevel coupled model transformations for precise and reusable definition of model behaviour. *J. Log. Algebr. Methods Program.*, 106:167–195, 2019.

[10] D. Méndez-Acuña, J. A. Galindo, T. Degueule, B. Combemale, and B. Baudry. Leveraging software product lines engineering in the development of external DSLs: A systematic literature review. *Comput. Lang. Syst. Struct.*, 46:206–235, 2016.

[11] J. E. Rivera, F. Durán, and A. Vallecillo. On the behavioral semantics of real-time domain specific visual languages. In P. C. Ölveczky, ed., *Revised Selected Papers from 8th Int. Wksh. on Rewriting Logic and Its Applications, WRLA 2010*, v. 6381 of *Lect. Notes in Comput. Sci.*, pp. 174–190. Springer, 2010.

[12] A. Rodríguez, F. Durán, A. Rutle, and L. M. Kristensen. Executing multilevel domain-specific models in Maude. *J. Object Technol.*, 18(2), article 4, 2019.

[13] A. Rodríguez, A. Rutle, L. M. Kristensen, and F. Durán. A foundation for the composition of multilevel domain-specific languages. In *Companion of 22nd ACM/IEEE Int. Conf. Model Driven Engineering Languages and Systems, MODELS Companion 2019*, pages 88–97. IEEE 2019.

[14] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework.* Pearson Education, 2008.

[15] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *IEEE Softw.*, 31(3):79–85, 2014.

Profunctor Optics, a Categorical Update (Extended Abstract)

Mario Román[1], Bryce Clarke[2], Derek Elkins[3], Jeremy Gibbons[1], Bartosz Milewski[4], Fosco Loregian[5], and Emily Pillmore[3]

[1] Dept. of Computer Science, University of Oxford, United Kingdom
jeremy.gibbons@cs.ox.ac.uk, mario.romangarcia@maths.ox.ac.uk
[2] Macquarie University, Sydney, Australia, bryce.clarke@mq.edu.au
[3] Independent researcher, derek.a.elkins@gmail.com, emilypi@cohomolo.gy
[4] Programming Cafe, bartosz@relisoft.com
[5] Centre for Mathematics, University of Coimbra, Portugal, fosco.loregian@gmail.com

Abstract

Profunctor optics [PGW17, BG18] are a family of composable bidirectional data accessors. They provide a powerful abstraction over many data transformation patterns in functional programming described in libraries such as Kmett's *lens* library [Kme18]. Generalizing a result by Pastro and Street [PS08], we get a new proof of the equivalence between existential and profunctor representations of the optics. This extends to the case of *mixed optics* proposed by Riley [Ril18]. We collect derivations from the existential to the concrete form, including many original ones. In particular, we present an elementary derivation of the optic known as *traversal*, solving a problem posed by Milewski [Mil17]. We discuss a novel approach to composition of optics, based on both distributive laws and coproducts of monads. This is work in progress.

1 Optics

In functional programming, *optics* are a modular representation of bidirectional data accessors. Boisseau and Gibbons' profunctor representation theorem [BG18] proves that they can be equivalently written as functions polymorphic over profunctors. This profunctor representation is convenient because it turns composition of optics into ordinary function composition.

Example 1.1. *Lenses* are type-changing getter/setter pairs, defined as $\mathbf{Lens}((A,B),(S,T)) := (S \to A) \times (S \times B \to T)$ for any four types A,B,S,T. *Prisms* are data accessors enabling alternatives, defined as $\mathbf{Prism}((A,B),(S,T)) = (S \to A + T) \times (B \to T)$ for any four types A,B,S,T. Both are optics, in the sense of the following Definition 1.2, which means they can be written in profunctor form, thanks to Theorem 1.3, and composed using ordinary function composition. The following code uses a *prism* (`postal`) to parse a string into a *postal address*. The prism is then composed with a lens that accesses the *street* subfield inside the *postal address* (`street`). The composite optic can view and update the *street* field inside the string.

```
let address = "45 Banbury Rd, OX1 3QD, Oxford"
address^.postal
  -- {Street: "45 Banbury Rd", Code: "OX1 3QD", City: "Oxford"}
address^.postal.street
  -- "45 Banbury Rd"
address^.postal.street %~ "7 Banbury Rd"
  -- "7 Banbury Rd, OX1 3QD, Oxford"
```

A first unified definition of *optic* was proposed by Milewski [Mil17], who also suggested its monoidal constraints. This definition has been presented using submonoids of endofunctors by Boisseau and Gibbons [BG18] and using monoidal actions by Riley [Ril18]. We extend this definition to what Riley [Ril18] suggested to call *mixed optics*; we also extend his proof to show that mixed optics for a pair of monoidal actions are morphisms defining a category. The main proof technique is (co)end calculus as described, for instance, by Loregian [Lor15].

Definition 1.2. Let $\mathbf{M}$ be a monoidal category and let $\mathbf{C}$ and $\mathbf{D}$ be two arbitrary categories. Let $(\underline{\ }) \colon \mathbf{M} \to [\mathbf{C}, \mathbf{C}]$ and $(\underline{\ }) \colon \mathbf{M} \to [\mathbf{D}, \mathbf{D}]$ be two strong monoidal functors. An **optic** from $(S, T) \in \mathbf{C} \times \mathbf{D}$ with *focus* on $(A, B) \in \mathbf{C} \times \mathbf{D}$ is an element of the coend

$$\mathbf{Optic}\left((A, B), (S, T)\right) := \int^{M \in \mathbf{M}} \mathbf{C}(S, \underline{M}A) \times \mathbf{D}(\underline{M}B, T),$$

For this extended definition, we present an analogue of Boisseau and Gibbons' profunctor representation theorem [BG18]. Its proof is based on Pastro and Street's study of *doubles* for monoidal categories [PS08]; however, it generalizes tensor products to arbitrary monoidal actions over two different categories.

Theorem 1.3 (Profunctor representation theorem, after Boisseau and Gibbons [BG18])**.** *In the conditions of Definition 1.2,*

$$\int_{P \in \text{た}} \mathbf{Sets}(P(A, B), P(S, T)) \cong \mathbf{Optic}((A, B), (S, T)),$$

where た *is the category of Tambara modules for the actions of* $\mathbf{M}$*.*

Our definition of *Tambara module* generalizes Tambara's original one [Tam06] to arbitrary pairs of monoidal actions. As Pastro and Street [PS08] showed for the original case, they can be equivalently described by as coalgebras for a comonad $\Theta \colon \mathbf{Prof}(\mathbf{C}, \mathbf{D}) \to \mathbf{Prof}(\mathbf{C}, \mathbf{D})$ defined on profunctors by

$$\Theta P(A, B) = \int_{M \in \mathbf{M}} P(\underline{M}A, \underline{M}B).$$

As a corollary, the category **Optic** is shown to be the full subcategory on representable profunctors of the co-Kleisli category of Θ. This opens the possibility of exploring two different ways of composing optics of different *families*. The first is to consider distributive laws between Pastro-Street comonads; the second is to consider products of comonads. We show that both, under suitable considerations, produce again Pastro-Street comonads. This technique can be used to get some optics present in the literature such as the *affine traversal* [PGW17], but also to produce some original ones. Composition of optics of different kinds is common in programming practice; but a justification of its correctness was missing from the literature.

2 Examples of optics

An important justification of Definition 1.2 is that it captures the common examples of optics that occur in programming. Milewski [Mil17] showed that *lenses*, *prisms* and *grates* fit the definition relying only on elementary applications of the Yoneda lemma. Boisseau and Gibbons [BG18] and then Riley [Ril18] have shown the same for other common optics. We address the problem of finding an elementary derivation of the *traversal*, as proposed by Milewski [Mil17]. A **traversal** from (S, T) with focus on (A, B) is an element of $\mathbf{C}\left(S, \sum_n A^n \times (B^n \to T)\right)$. We describe *traversals* as the optic for *power series functors*, also called polynomial functors in one variable [Koc09]. This is related to the more common description of traversals as optics for *traversable functors* by a result of Jaskelioff and O'Connor [JO15] that characterizes traversables as coalgebras for a certain parameterized comonad.

Proposition 2.1. *Given some functor* $C \in [\mathbb{N}, \mathbf{C}]$ *from the discrete category of the natural numbers, we can define a* power series *functor* $F \colon \mathbf{C} \to \mathbf{C}$ *given by* $F(A) = \sum_{n\in\mathbb{N}} C_n \times A^n$*. This induces a monoidal action that we call* $\mathrm{Series} \colon [\mathbb{N}, \mathbf{C}] \to [\mathbf{C}, \mathbf{C}]$. Traversals *are optics for this action* $\mathrm{Series} \colon [\mathbb{N}, \mathbf{C}] \to [\mathbf{C}, \mathbf{C}]$.

Proof. Unfolding the definitions, we want to prove that

$$\int^{C\in[\mathbb{N},\mathbf{C}]} \mathbf{C}\left(S, \sum_{n\in\mathbb{N}} C_n \times A^n\right) \times \mathbf{C}\left(\sum_{n\in\mathbb{N}} C_n \times B^n, T\right) \cong \mathbf{C}\left(S, \sum_{n\in\mathbb{N}} A^n \times (B^n \to T)\right).$$

The fact that there exists an isomorphism between the two sets, natural in A, B, S and T, is a consequence of continuity of the hom-functor and the Yoneda lemma. □

We collect novel derivations for many other optics. The following are some of them, together with their generating monoidal actions.

Name	Monoidal action	Concrete form
Glass	Product and exponential	$((S \to A) \to B) \to S \to T$
Unsorted Traversal	Combinatorial species	$S \to \int^{n\in\mathbf{B}} A^n \times (B^n \to T)$
Algebraic lens	Product by a ψ-algebra	$(S \to A) \times (\psi S \times B \to T)$
Kaleidoscope	Applicative functors	$\prod_{n\in\mathbb{N}}(A^n \to B) \to (S^n \to T)$

The generalization to *mixed optics* allows us to consider *degenerate optics*. These are optics where one of the categories is the terminal category. Degenerate optics include *getters*, *setters* and *folds*; as they appear in Kmett's *lens* library [Kme18]. This definition also captures some variants of lenses and, remarkably, a generalization of lenses to an arbitrary monoidal category proposed by Myers and Spivak [Spi19, §2.2].

3 A case study

Let us discuss an example of how our results can be used in practice. Consider the `iris` dataset [Fis36], where each entry represents a *flower* described by its species and four real number measurements.

Example 3.1. An *algebraic lens* (`measurements`) for the list monad is used first as an ordinary lens to access the first element of the dataset (line 1), and then to encapsulate some learning algorithm that classifies measurements into a species (line 2). Consider a *kaleidoscope* that extends an aggregating function on the reals to the measurements (`aggregateWith`). Our work has shown that both fit Definition 1.2, which allows us to use Theorem 1.3 and our results on composition of optics to join them into a new kaleidoscope (`measurements.aggregateWith`, in line 3). It is remarkable that we just use ordinary function composition.

```
(iris !! 1)^.measurements
-- (4.9, 3.0, 1.4, 0.2)
iris ?. measurements (Measurements 4.8 3.1 1.5 0.1)
-- Iris Setosa (4.8, 3.1, 1.5, 0.1)
iris >- measurements.aggregateWith mean
-- Iris Versicolor (5.8, 3.0, 3.7, 1.1)
```

References

[BG18] Guillaume Boisseau and Jeremy Gibbons. What you needa know about Yoneda: Profunctor optics and the Yoneda lemma (functional pearl). *Proc. ACM Program. Lang.*, 2(ICFP):84:1–84:27, 2018.

[Fis36] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Ann. Eugenics*, 7(2):179–188, 1936.

[JO15] Mauro Jaskelioff and Russell O'Connor. A representation theorem for second-order functionals. *J. Funct. Program.*, 25, article e13, 2015.

[Kme18] Edward Kmett. lens library, version 4.16. 2018. https://hackage.haskell.org/package/lens-4.16

[Koc09] Joachim Kock. Notes on polynomial functors. Manuscript, version 2009-08-05, 2009. Available at http://mat.uab.es/~kock/cat/polynomial.pdf

[Lor15] Fosco Loregian. This is the (co)end, my only (co)friend. arXiv preprint arXiv:1501.02503, 2015.

[Mil17] Bartosz Milewski. Profunctor optics: the categorical view. 2017. Available at https://bartoszmilewski.com/2017/07/07/profunctor-optics-the-categorical-view/

[PGW17] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *Programming J.*, 1(2), article 7, 2017.

[PS08] Craig Pastro and Ross Street. Doubles for monoidal categories. *Theory Appl. Categ.*, 21(4):61–75, 2008.

[Ril18] Mitchell Riley. Categories of optics. arXiv preprint arXiv:1809.00738, 2018.

[Spi19] David I. Spivak. Generalized lens categories via functors $\mathcal{C}^{\mathrm{op}} \to \mathsf{Cat}$. arXiv preprint arXiv:1908.02202, 2019.

[Tam06] Daisuke Tambara. Distributors on a tensor category. *Hokkaido Math. J.*, 35(2):379–425, 2006.

Algebra-Oriented Proofs for Optimisation of Lockset Data Race Detectors[*]

Justus Sagemüller, Volker Stolz, and Olivier Verdier

Western Norway University of Applied Sciences, Bergen, Norway
{jsag,vsto,over}@hvl.no

Abstract

Dynamic detectors for potential data races, based on lockset-intersection tracking, are widely used to safety-check concurrent programs. But to make this real-world-feasible, optimisations can be necessary whose validity has so far lacked a solid proof foundation.

We demonstrate that lockset algorithms possess a simple algebraic structure (a commutative monoid), which can simplify the formalisation of optimisations in a proof assistant.

Concurrent programs require synchronisation mechanisms if shared mutable data is to be used safely. Specifically, unrestricted concurrent mutation easily leads to data races [4], causing data corruption which can be disastrous yet hard to detect.

Besides approaches such as Software Transactional Memory, the standard mechanism to avoid this is using *locks*, which can prevent one thread from proceeding until it is safe to do so. This can cause its own problems though, in the most extreme *deadlock* [1] but also simply missed parallelism opportunities. It is thus desirable to minimise the use of locks, but the nondeterministic nature of race conditions requires reliably detecting them, to ultimately avoid them from being possible in production code.

Such detectors come in both static (compile-time) and dynamic (runtime) flavours. We focus on *lockset algorithms* as defined in [5] (in the following called *Eraser*), specifically the *simple lockset algorithm*.[1] The way this is usually presented is as a state machine on top of the running program: for each thread t, the set of currently-held locks is updated as locks are taken or unlocked on that thread. Furthermore, for each shared variable v, a lockset $C(v)$ is kept as the *protecting set*. This starts out as the set of all possible locks, and whenever thread t accesses the variable, it is intersected with the set of lock held by that thread, i.e.

$$C(v) \leftarrow C(v) \cap \mathrm{locks\_held}(t).$$

If this intersection ever becomes empty, it means a thread has accessed the variable while not holding any lock that other threads have held to ensure sovereignty over the variable when accessing it, so the empty set indicates a potential race condition.

Since the detector does not affect the program flow itself, it can (instead of as a simultaneously-running state machine) as well be formulated to process only a *trace* of the program run[3], where "trace" can be understood as just a stream/list of actions

$$\mathrm{Op} = \begin{cases} \mathrm{Lock}\ (t : \mathrm{ThreadId})\ (l : \mathrm{LockId}) \\ \mathrm{Unlock}\ (t : \mathrm{ThreadId})\ (l : \mathrm{LockId}) \\ \mathrm{Access}\ (t : \mathrm{ThreadId})\ (v : \mathrm{VarId}) \end{cases}$$

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

[1] The actual Eraser algorithm has additional states, but these are mostly relevant for initialisation.

Algebraic formulation. For the actual results of the algorithm (i.e. the protecting sets and specifically if they become empty), it is not relevant to consider individual locking or unlocking operations. Rather, it is sufficient to keep track of only the actual lockset states, and even that only at the times when a variable is accessed (because that will be the set which is intersected with the protecting set). So instead of a trace of events, one can consider a trace of accesses together with the lock states: for example, a trace for one thread, one variable, and two locks, is compressed thus:

$$\begin{tikzcd}
& \mathrm{L}^{t_0}_{l_0} \arrow[rr] \arrow[d, dash, dashed] & & \mathrm{L}^{t_0}_{l_1} \arrow[r] \arrow[d, dash, dashed] & \mathrm{A}^{t_0}_{v_0} \arrow[r] \arrow[d, dash, dashed] & \mathrm{U}^{t_0}_{l_0} \arrow[rr] \arrow[d, dash, dashed] & & \mathrm{L}^{t_0}_{l_0} \arrow[rr] \arrow[d, dash, dashed] & & \mathrm{U}^{t_0}_{l_1} \arrow[r] \arrow[d, dash, dashed] & \mathrm{A}^{t_0}_{v_0} \arrow[d, dash, dashed] \\
\tbinom{0}{0} \arrow[rr] \arrow[dr, dash, dashed] & {} & \tbinom{1}{0} \arrow[rr] & {} & (\tbinom{1}{1}, \mathrm{A}^{t_0}_{v_0}) \arrow[rr] \arrow[d, dash, dashed] & {} & \tbinom{0}{1} \arrow[rr] & {} & \tbinom{1}{1} \arrow[rr] & {} & (\tbinom{1}{0}, \mathrm{A}^{t_0}_{v_0}) \arrow[d, dash, dashed] \\
& {} \arrow[rrr] & & & (\tbinom{1}{1}, \mathrm{A}^{t_0}_{v_0}) \arrow[rrrrrr] & & & & & & (\tbinom{1}{0}, \mathrm{A}^{t_0}_{v_0}).
\end{tikzcd}$$

In this diagram, arrows in the top line represent sequencing of the original events in the trace. In the bottom line, the arrows represent only the transitions between states that actually need to be considered for the race-condition checking.

The vector notation used here already suggests that it is not really necessary to consider the states as sets. The intersection operation that the lockset algorithm applies at each access is but a special case of a *pointwise multiplication*. So generically, these sets can be seen as an instance of an algebra or more precisely a *ring*. Considering only the intersection operation, it is a *monoid*, with the universal set (or, vector-of-all-1-s) as unit. This operation will in the following be written simply as juxtaposition, i.e. $s_a s_b \equiv s_a \cap s_b$.

What the Eraser algorithm does is then left-scanning over the accesses and always applying the held lockset to the right of the protecting set, with the monoid operation:

$$\to (\tbinom{1}{1}, s_0 \tbinom{1}{1}) \to (\tbinom{1}{0}, s_0 \tbinom{1}{1} \tbinom{1}{0})$$

so starting from $s_0 = \tbinom{1}{1}$, the final state would be

$$\tbinom{1}{1}\tbinom{1}{1}\tbinom{1}{0} = \tbinom{1}{1}\tbinom{1}{0} = \tbinom{1}{0}.$$

For arbitrary locking and access operations, the sequence of relevant checking states becomes

$$(m_0, s_0 m_0) \to (m_1, s_0 m_0 m_1) \to (m_2, s_0 m_0 m_1 m_2) \to \ldots$$

In the general setting of multiple threads, locks and variables, there needs to be one lockset for each thread and one protecting set for each variable, so e.g.

$$\to \big((m_{0,0}, m_{0,1}, \ldots), (\mathrm{A}^{t_1}_{v_0})\big) \to \big((m_{1,0}, m_{1,1}, \ldots), (\mathrm{A}^{t_0}_{v_1})\big) \to \big((m_{2,0}, m_{2,1}, \ldots), (\mathrm{A}^{t_1}_{v_1})\big)$$

is checked as

$$\left((m_{0,0} \ldots), \begin{pmatrix} s_{0,0} m_{0,1} \\ s_{0,1} \\ \vdots \end{pmatrix} \right) \to \left((m_{1,0} \ldots), \begin{pmatrix} s_0 m_{0,1} \\ s_{0,1} m_{1,0} \\ \vdots \end{pmatrix} \right) \to \left((m_{2,0} \ldots), \begin{pmatrix} s_0 m_{0,1} \\ s_{0,1} m_{1,0} m_{2,1} \\ \vdots \end{pmatrix} \right).$$

Effectively, after an access to a variable v, its protecting state is a fold over all previous accesses to v of the lockset state at that time of the corresponding thread.

Computer-assisted proofs. The folding operation lends itself well to a purely-functional implementation, which can be in a language such as Coq:

```
Axiom drcAlgebraic : forall tr v t, let trx := tr ++ [Access t v]
  in drc trx
     <-> match compressTr trx with
         | inl cTr => fold_left pr (map (fun q => fst q (fst (snd q)))
                                        (filter (VId.eqb v ∘ snd ∘ snd) cTr)
                                   ) I <> O
         | inr _ => False
         end.
```

Here, `drc` is an abstract notion of what a data-race checker on a trace of events is. The axiom states that it should be equivalent to our algebraic notion, where `I` is the all-ones state and `O` an all-zeroes state (representing the empty protecting set, i.e. a potential data race). The algebraic form works on the `compressTr` form of the trace. This is actually not defined for arbitrary traces of operations, but only for *well-formed* traces.[2]

We propose using this to prove the correctness of commonly used optimizations that compress or elide information from the trace only based on the underlying monoid. An advantage of the algebraic viewpoint is that it decouples the correctness proofs from any concrete data structure implementing Eraser, as long as an implementation (e.g. through vectors) has the required algebraic properties. For example, we show that repetition of well-formed blocks with balanced locks and unlocks does not add new knowledge about data races, and can hence be elided from the trace:[3]

```
Lemma ast_balanced: forall (pre tr post: list op),
      balanced tr -> wf (pre++tr++post)
   -> forall n, n > 0 -> drc (pre++tr++post) <-> drc (pre++(repeat tr n)++post).
```

This and other equivalences can then be used to avoid the placement of redundant instrumentation (for related work see e.g. [2]) into a system-under-test, and hence through reduced event generation positively affect the computation time for dynamic data race checking.

References

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

[2] C. Flanagan and S. N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In G. Castagna, editor, *Proc. of 27th European Conf. on Object-Oriented Programming, ECOOP 2013*, volume 7920 of *Lecture Notes in Computer Science*, pages 255–280. Springer, 2013.

[3] S. Jakšić, D. Li, K. I. Pun, and V. Stolz. Stream-based dynamic data race detection. In *Norsk Informatikkonferanse, NIK 2018*, 12 pp. 2018. Available at http://ojs.bibsys.no/index.php/NIK/article/view/511.

[4] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, 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.

[2] A well-formed trace is one that could actually happen in a lock-obeying run of a program. It can not have the same lock taken simultaneously by different threads (nor a lock taken more than once by a single thread).

[3] Without the `post`, `drc` ↔ `drc` is not exactly the same as equivalent lock-info.

Study of Recursion Elimination for a Class of Semi-Interpreted Recursive Program Schemata

Nikolay V. Shilov

Innopolis University, Innopolis, Republic of Tatarstan, Russia
shiloviis@mail.ru

Abstract

We study templates (i.e. control flow structures with uninterpreted functional and predicate symbols commonly known as program schemata) for *descending* and *ascending* dynamic programming, discuss these templates from programming theory perspective in terms of translation of recursive program schemata to iterative ones with or without dynamic memory, suggested sufficient conditions when the recursive template can be translated into iterative program schemata with fix-size static memory.

More than 50 years passed since the "Golden Age" of Theory of Program Schemata in 1960-70's. Great computer scientists contributed to these studies: John McCarthy, Edsger Dijkstra, Donald Knuth, Amir Pnueli... Studies of `go-to` elimination (structured program Böhm-Jacopini theorem about a translation of spaghetti-like iterative code to more understandable and easier to verify iterative code) and recursion elimination (i.e. how to translate recursive program schemata and programs to iterative ones) were very popular in 1960-1970's [4]. Recursion elimination was very popular because it is about translation from easier to design and verify declarative code to more efficient imperative code. Many fascinating examples of recursion elimination have been examined [3, 2, 7] (e.g. *tail-recursion* that is basically a recursive variant of `go-to`). In the paper we study a recursion pattern that doesn't match the tail-recursion, but matches well the pattern of Bellman equation, a general form for recursive dynamic programming. We study this pattern of recursive dynamic programming as a template (i.e. uninterpreted or semi-interpreted program scheme with a variable arity of symbols/functions/predicates) [6], discuss sufficient conditions for the interpretation of functional and predicate symbols when the recursive scheme may be translated to iterative schemata with (i) an associative array with a pre-computed size, (ii) an integer array with pre-computed size, and (iii) a fix-size static memory.

Dynamic Programming was introduced by Richard Bellman in the 1950s to tackle optimal planning problems. *Bellman equation* is a name for recursive functional equality for the objective function that expresses the optimal solution at the "current" state in terms of optimal solutions at next (changed) states, it formalizes a so-called *Bellman Principle of Optimality*: *an optimal program (or plan) remains optimal at every stage.* In the present paper we study a class of Bellman equations that matches the following recursive pattern:

$$G(x) \ = \ if\ p(x)\ then\ f(x)\ elseg\Big(x,\ \Big\{h_i\big(x, G(t_i(x))\big),\ i \in [1..n(x)]\Big\}\Big) \tag{1}$$

We consider the pattern as a *recursive program scheme* (or template) [6], i.e. a recursive control flow structure with *uninterpreted symbols*:

- G is the *main* functional symbol representing (after interpretation of *ground* functional and predicate symbol) the objective function $G : X \to Y$ for some X and Y;

- p is a ground predicate symbol representing (after interpretation) some *known*[1] predicate $p \subseteq X$;
- f is a ground functional symbol representing (after interpretation) some known[1] function $f : X \to Y$;
- g is a ground functional symbol representing (after interpretation) some known[1] function $g : X \times Z^* \to X$ for some appropriate Z (with a variable arity $n(x) : X \to \mathbb{N}$);
- all h_i and t_i $(i \in [1..n(x)])$ are ground functional symbols representing (after interpretation) some known[1] function $h_i : X \times Y \to Z$, $t_i : X \to X$ $(i \in [1..n(x)])$.

In the sequel we do not make an explicit distinction in notation for symbols and interpreted symbols but just verbal distinction by saying, for example, *symbol* g and *function* g.

Let us consider a function $G : X \to Y$ that is defined by the interpreted recursive scheme (1). Let us define two sets $bas(v), spp(v) \subseteq X$:

- base $bas(v) = \ if\ p(v)\ then\ \emptyset\ else\ \{t_i(v) : i \in [1..n(v)]\} \subseteq X$ comprises all values that are immediately needed to compute $G(v)$;
- support $spp(v)$ is the set of all values that appear in the call-tree of $G(v)$.

Note that $bas(v)$ is always finite and if G is defined on v then the support $spp(v)$ is finite. When $G(v)$ is defined, the support can be computed by the following algorithm:

$$spp(x) \ = \ if\ p(x)\ then\ \{x\}\ else\ \{x\} \cup (\bigcup_{y \in bas(x)} spp(y)). \tag{2}$$

Let us specify and verify the following *iterative template for/of (ascending) dynamic programming*:

- Template Applicability Conditions TAC:
  1. I is an interpretation for ground symbols in the scheme (1);
  2. $n : X \to \mathbb{N}$ is the arity function of interpreted g;
  3. $G : X \to Y$ is the objective function, i.e. a solution of the interpreted scheme (1);
  4. $t_1, \ldots t_n : X \to X$ are functions that computes the base;
  5. $spp : X \to 2^X$ is the support function for G;
  6. $NiX \notin X$ is a distinguishable fixed indefinite value[2] for X;
- Template Pseudo-Code TPC:
  1. $VAR\ LUT\ \ : assosiative\ array\ indexed\ by\ spp(v)\ with\ values\ in\ Y$;
  2. $LUT := \ array\ filled\ by\ NiX$;
  3. $for\ all\ u \in spp(v)\ do\ if\ p(u)\ then\ LUT[u] := f(u)$;

[1] i.e. that we know how to compute

[2] NiX — *Not in* X, similarly to *Non a Number* — `NaN`.

```
4. $while\ LUT[v] = NiX\ do$
   $let\ u \in spp(v)\ be\ any\ element\ in\ spp(v)$
       $such\ that\ LUT[u] = NiX\ and$
       $LUT[t_i(u)] \neq NiX\ for\ all\ i \in [1..n(u)]$
   $in\ LUT[u] :=\ g\Big(u,\ \Big\{h_i\big(u, LUT[t_i(u)]\big),\ i \in [1..n(u)]\Big\}\Big).$
```

Note that the template is not a *standard program scheme*, but a scheme augmented by *associative array* (namely LUT).

Proposition 1. *Assuming TAC, the following holds for every $v \in X$:*

1. *if $G(v)$ is defined then interpreted template TPC terminates after $|spp(v)|$ iterations of both loops, and $LUT[v] = G(v)$ by termination;*
2. *if $G(v)$ is not defined then interpreted template TPC never terminates.*

The advantage of TPC is the use of an associative array that is allocated once instead of a stack, which is required to translate general recursion. Nevertheless, a natural question arises: is a finite static memory sufficient when computing this function? Unfortunately, in general, the answer is no according to the following proposition by M.S. Paterson and C.T. Hewitt [6].

Proposition 2. *The following special case of the recursive template (1)*

$$F(x) =\ if\ p(x)\ then\ x\ else\ f(F(g(x)), F(h(x)))$$

is not equivalent to any standard program scheme (i.e. an uninterpreted iterative program scheme with finite static memory).

Proposition does not imply that dynamic memory is *always* required; it just says that for *some* interpretations of *uninterpreted* symbols p, f, g and h the size of required memory depends on the input data. But if p, f, g and h are *interpreted*, it may happen that function F can be computed by an iterative program with a finite static memory. For example, Fibonacci numbers

$$Fib(n) =\ if\ (n = 0\ or\ n = 1)\ then\ 1\ else\ Fib(n-2) + Fib(n-1)$$

matches the pattern of the scheme in the above proposition 2, but just three integer variables suffice to compute it by an iterative program.

The following proposition states sufficient conditions when a finite static memory suffices to compute the recursive function (1).

Proposition 3. *Assume that TAC holds altogether with the following additional conditions:*

- *arity function $n : X \to \mathbb{N}$ is some constant $n \in \mathbb{N}$;*
- *base functions $t_1, \dots\ t_n$ are interpreted in such a way that t_1 is invertible and $t_i = (t_1)^i$ for all $i \in [1..n]$;*
- *interpreted predicate p is t_1-closed in the following sense: $p(u) \Rightarrow p(t_1(u))$ for all $u \in X$.*

Let $m \in \mathbb{N}$ be number of static variables that suffice to implement imperative iterative algorithms to compute interpreted ground predicate and functions p, f, h_i ($i \in [1..n]$), t_1 and t_1^- for any input value. Then the objective function G may be computed by an imperative iterative algorithm with $2n + m + 2$ static variables.

(Let us skip a proof of the statement because of space limitations.)

To the best of our knowledge, use of integer arrays for efficient translation of recursive functions of integer argument was suggested first in [1]. In the cited paper this technique of recursion implementation was called *production mechanism.* The essence of the production mechanism consists in support evaluation (that is a set of integers), array declaration with a proper index range, and fill-in this array in bottom-up (i.e. ascending) manner by values of the objective function. Use of auxiliary array was studied also in [5]. The book [5] doesn't use templates but translation techniques asymptotically but is more space efficient that our approach. (For example, if to use techniques from [5], then the length of the longest common subsequence can be computed in linear space, while our approach needs a quadratic space.)

Nevertheless, a novelty of our study consists in the use of templates (understood as semi-interpreted program schemata) and sematic sufficient conditions that allow recursive programs to be computed efficiently by iterative imperative programs (with either an associative array or just with a finite fixed size static memory).

Acknowledgment

The author thanks the anonymous reviewers and Dr. Daniel De Carvalho for significant comments both in the content and in the language of the article.

References

[1] G. Berry. Bottom-up computation of recursive programs. *Theor. Inf. Appl.*, 10(3):47–82, 1976.

[2] R. S. Bird. Zippy tabulations of recursive functions. In P. Audebaud and C. Paulin-Mohring, editors, *Proc. of 9th Int. Conf. on Mathematics of Program Construction, MPC 2008*, volume 5133 of *Lecture Notes in Computer Science*, pages 92–109. Springer, 2008.

[3] J. Cowles and R. Gamboa. Contributions to the theory of tail recursive functions. 2004. Available at http://www.cs.uwyo.edu/~ruben/static/pdf/tailrec.pdf.

[4] D.E. Knuth. Textbook examples of recursion. arXiv preprint cs/9301113, 1991. Available at https://arxiv.org/abs/cs/9301113.

[5] Y. A. Liu. *Systematic Program Design: From Clarity to Efficiency.* Cambridge University Press, 2013.

[6] M.S. Paterson and C.T. Hewitt. Comperative schematology. In *Proc. of ACM Conf. on Concurrent Systems and Parallel Computation*, pages 119–127. ACM, 1970.

[7] N.V. Shilov. Etude on recursion elimination. *Modeling and Analysis of Information Systems*, 25(5):549–560, 2018.

Formal Verification of Maritime Autonomous Systems Using UPPAAL STRATEGO

Fatima Shokri-Manninen[1],
Jüri Vain[2], and Marina Waldén[1]

[1] Dept. of Information Technologies, Åbo Akademi University, Finland
{fatemeh.shokri,marina.walden}@abo.fi,
[2] Dept. of Software Science, Tallinn University of Technology, Estonia
{juri.vain}@taltech.ee

Abstract

Lately the demand for autonomous ships has grown substantially. Autonomous ships are expected to navigate safely and avoid collisions following accepted navigation rules. We model the autonomous system as a Stochastic Priced Timed Game using UPPAAL STRATEGO. The behaviour of the controller is optimised and verified in order to achieve the goal to safely reach the destination at a minimum cost.

1 Introduction

The demand for unmanned ships has risen aiming at reducing operation costs due to minimal crew on board and safety at sea but also promoting remote work. Autonomous ships are expected to make more and more decisions based on their current situation at sea without direct human supervision. This means that an autonomous ship should be able to detect other vessels and make appropriate adjustments to avoid collision by maintaining maritime traffic rules. However, the existence of a 'virtual captain' from the shore control centre (SCC) is still a must to perform critical or difficult operations [1]. The presence of virtual captains also increase the chances of spotting a cyber-attacks [10]. The connectivity between ships and SCC has to guarantee sufficient communication for sensor monitoring and remote control [5] when SCC intervention is needed. This connectivity also plays an important role for the safety of operations concerning collision avoidance in the remote-controlled scenarios as it needs to be fast for transforming the data and receiving information regarding the decision from SCC. For preventing collisions at sea, the International Maritime Organization (IMO) [6] published navigation rules to be followed by ships and other vessels at sea which are called Convention On the International Regulations (COLREG).

When developing the autonomous ship navigation system, quality assurance via model-based control synthesis and verification is of utmost importance. UPPAAL STRATEGO [4] is a branch of the UPPAAL [2] family of model checker tools. It uses machine learning and model checking techniques to synthesize optimal control strategies. Hence, it is a good candidate for control synthesis tool which satisfies above mentioned requirements.

In our research, we aim at adapting formal modelling with UPPAAL STRATEGO for verifying and synthesizing safe navigation of autonomous ships. As an additional contribution, we improve the autonomous ships navigation performance regarding its safety and security at the same time planning for optimal route and scheduling maneuvers according to COLREG rules. Furthermore, this study relies on experience reports regarding the identified challenges for formal modelling of autonomous systems.

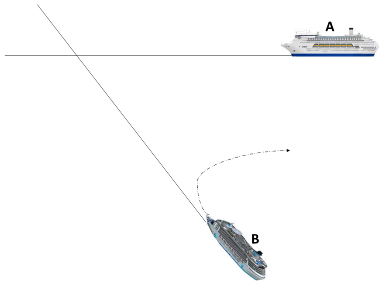

Figure 1: Autonomous Navigation of Ships

2 Related work

There have been a variety of studies on autonomous ship navigation obeying COLREGS rules. Among these fuzzy logic [8], interval programming [3], and 2D grid map [11] could be mentioned. However, the previous approaches do not deal with verification for modelling safe navigation. Moreover, non-deterministic behaviour of autonomous ship, communication delays, sensor failure and weather feature are not considered in their models.

Recently, in MAXCMAS project [12], COLREGs rules have been implemented in collision avoidance module (CAM) software where generates collision avoidance decision and action as soon as collision risk is detected. In spite of their various simulation tools, verification methods are discussed only implicitly.

UPPAAL STRATEGO has proven its relevance in several case studies, where optimal strategies have been through statistical model checking and machine learning. Examples include for instance adaptive cruise control [7].

3 Model

When modelling navigation manoeuvres of autonomous ships, we start with standard situations, addressed in COLREG. As an example, let us consider a scenario where two ships have intersecting courses as depicted in Figure 1.

In this example, in spite of the existence of monitor from the SCC, we assume also that ships are equipped with intelligent controllers. According to Rule 15 of COLREG [9]; when two power driven vessels have intersecting courses with the risk of collision, the vessel which has the other on her own starboard (right) side shall keep out of the way and avoid crossing ahead of the other vessel. If there is a risk of collision between vessels headed for a crossing situation, a vessel has to give way to the vessel on its starboard side. In this case the vessel giving way should adjust its speed and/or course to pass behind the incoming vessel. The adjustment will therefore be made to the starboard side. In the case depicted in Figure 1, Ship B should give way while ship A maintains its direction and speed.

The controller of ship B has a choice to slow down its speed instead of altering its path to pass ship A. By doing this, the expected arrival time might not be as late as when following a redirected route. However, if for some reason ship A is slowing down, then the controller should navigate ship B safely to another route.

We model the system as a Stochastic Priced Timed Game using the tool UPPAAL STRATEGO where the controller of ship B should dynamically plan its maneuver, while the opponent (ship A) moves according to its preset trajectory forces ship B to change its route. In this game, we define the fuel consumption (FC) as a the price to be minimized under the safe strategy. The change in velocity of the ship is directly related to FC, so that the consumption of fuel increases if the ship slows down or speeds up rather than changes the route, causing the price to increase.

The goal is that the ships move to their target positions in a safe way (without the risk of a collision) while at the same time optimizing the travel times and also the fuel consumption. We rely on reinforcement learning and Q-learning supported via UPPAAL STRATEGO to optimize and verify the behaviour of the controller in order to achieve the goal.

4 Conclusion

In this paper, the approach for the control synthesis has been stated as a stochastic two players game with the goal of collision avoidance. Taking into account several practically important side constraints such as wind, currents, navigation mistakes by adversary's vessel, and involvement of other obstacles (nautical signs, small boats) complicates the synthesis task and presumes the validation of the approach under extra constraints not studied in standard game-theoretic setting yet.

Acknowledgments

This study has been partly supported by the Academy of Finland project ESC (grant no. 308980) and the Estonian Ministry of Education and Research institutional research grant no. IUT33-13.

References

[1] Sauli Ahvenjärvi. The human element and autonomous ships. *TransNav: Int. J. on Marine Navigation and Safety of Sea Transportation*, 10(3):517–521, 2016.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Revised Lectures from Int. School on Formal Methods for the Design of Real-Time Systems, SFM-RT 2004*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[3] Michael R. Benjamin, Joseph A. Curcio, John J. Leonard, and Paul M. Newman. Navigation of unmanned marine vehicles in accordance with the rules of the road. In *Proc. of 2006 IEEE Int. Conf. on Robotics and Automation, ICRA 2006*, pages 3581–3587. IEEE, 2006.

[4] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Uppaal Stratego. In Christel Baier and Cesare Tinelli, editors, *Proc. of 21st Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2015*, volume 9035 of *Lect Notes in Computer Science*, pages 206–211. Springer, 2015.

[5] Marko Höyhtyä, Jyrki Huusko, Markku Kiviranta, Kenneth Solberg, and Juha Rokka. Connectivity for autonomous ships: Architecture, use cases, and research challenges. In *Proc. of 2017 Int. Conf. on Information and Communication Technology Convergence, ICTC 2017*, pages 345–350. IEEE, 2017.

[6] IMO. Convention on the international regulations for preventing collisions at sea (COLREGs), 1972.

[7] Kim Guldstrand Larsen, Marius Mikučionis, and Jakob Haahr Taankvist. Safe and optimal adaptive cruise control. In Roland Meyer, Andre Platzer, and Heike Wehrheim, editors, *Correct System*

Design: Proc. of Symp. in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, volume 9360 of *Lecture Notes in Computer Science*, pages 260–277. Springer, 2015.

[8] Sang-Min Lee, Kyung-Yub Kwon, and Joongseon Joh. A fuzzy logic for autonomous navigation of marine vehicles satisfying COLREG guidelines. *Int. J. Contr. Autom. Syst.*, 2(2):171–181, 2004.

[9] L.P. Perera, J.P. Carvalho, and C. Guedes Soares. Autonomous guidance and navigation based on the COLREGs rules and regulations of collision avoidance. In *Proc. of Int. Workshop on Advanced Ship Design for Pollution Prevention, ASDEPP 2009*, pages 205–216, 2009.

[10] Kimberly Tam and Kevin Jones. Cyber-risk assessment for autonomous ships. In *Proc. of 2018 Int. Conf. on Cyber Security and Protection of Digital Services, Cyber Security 2018*, 8 pp. IEEE, 2018.

[11] Ken Teo, Kai Wei Ong, and Hoe Chee Lai. Obstacle detection, avoidance and anti collision for Meredith AUV. In *Proc. of OCEANS 2009 MTS/IEEE Biloxi Conf.*, 10 pp. IEEE, 2009.

[12] J.M. Varas, S. Hirdaris, R. Smith, P. Scialla, W. Caharija, Z. Bhuiyan, T. Mills, W. Naeem, L. Hu, I. Renton, D. Motson, E. Rajabally. MAXCMAS project: Autonomous COLREGs compliant ship navigation. In *Proc. of 16th Conf. on Computer Applications and Information Technology in the Maritime Industries, COMPIT '17*, pages 454–464. Technische Univ. Hamburg, 2017.

B$^\sharp$: Enabling Reusable Theories in Event-B

James Snook, Thai Son Hoang, and Michael Butler

Electronics and Computer Science, University of Southampton, United Kingdom
{jhs1m15,t.s.hoang,mjb}@ecs.soton.ac.uk

1 Background

The Event-B formal method [1] is used for system modelling and verification, it is supported by the Rodin IDE (Integrated Development Environment) [2]. Event-B was designed for modelling discrete systems using a rich set theoretic modelling language; it was not designed as a general theorem prover. This design focus resulted in some useful structures being difficult to model in a reusable way. This was partially addressed by the introduction of the Theory Plug-in [5] which extended the Event-B language with polymorphic recursive datatypes (allowing types such as lists and the naturals to be easily created) and user-defined operators. User defined operators came in two flavours, axiomatically defined operators and constructively defined operators (working more like functions in other languages, substituting arguments into expressions).

The extended Event-B syntax has been used to construct axiomatic definitions of types, e.g., the real numbers have been modelled as a commutative field, which has been used within Event-B system modelling [4]. This approach results in a lot of repetition in definition and theorem proving. For example, when constructing a field it is not possible to reuse a group construct, which results in it being necessary to repeat the group axioms for addition and multiplication, and repeat any group theorems and proofs. This was noted in [9], which proceeded to show that abstract mathematical structures such as monoids could be created using the Event-B set syntax in such a way that they could be related to concrete types (such as the naturals). Further, the abstract types could be used to construct other abstract types (e.g., using the group definition to facilitate the construction of a field). There were several problems noted with this approach, such as proving a concrete type was an instance of an abstract type (e.g., that zero and addition form a monoid) did not automatically allow theorems to be used on the concrete type (the user needed to manually move the theorems). It was noted that the work-arounds to these problems were repetative enough to be done automatically, and could be applied during a translation from another language. The B$^\sharp$ language was proposed [10] for this purpose, with syntactic elements designed for the construction of mathematical types in such a way that they could be translated to the Event-B syntax, allowing them to be used by an Event-B modeller.

2 The B$^\sharp$ Tools

An initial implementation of the B$^\sharp$ language has been made. This is constructed in a way to make it compatible with the Rodin tool set. Several mathematical structures have been implemented using the B$^\sharp$ tool to test its effectiveness e.g.,

$$\begin{aligned}&\textbf{Class}\ Monoid[M] : SemiGroup\ (ident : M)\\ &\qquad \textbf{where}\ \forall x : M \cdot equ(op(x, ident), x) \land equ(op(ident, x), x)\{\ldots\}\end{aligned} \tag{1}$$

This declares that a *Monoid* is a *SemiGroup* (*SemiGroup* is a previously defined type with an equivalence relation *equ* and a binary operator *op*, and is referred to as the supertype of

$Monoid$) with an identity ($ident$). The **where** statement introduces a series of predicates to constrain the class, in this case they define the identity to work in the expected way. Within the type body ($\{\ldots\}$) theorems and functions are declared. For example the following theorem about the identity:

$$\forall x : M \cdot equ(op(x, ident), ident) \Leftrightarrow equ(x, ident) \tag{2}$$

M was declared above (1), and can be used as an instance of the Monoid class anywhere within the type body. As can the op, equ and $ident$ members of the $Monoid$, they do not need to be explicitly quantified over.

Concrete types are also constructed such as the natural numbers (Nat). The following statement is used to show that zero and addition form a commutative monoid:

$$\textbf{Instance}\ CommMonoid\langle pNat\rangle\ (add, zero)\ addMon \tag{3}$$

$addMon$ is a name allowing the $CommMonoid$ to be explicitly referenced in other B$^\sharp$ statements.

The B$^\sharp$ tool uses an instance statement like (3) to produce an Event-B theorem stating that add and $zero$ form a $CommMonoid$. It also instantiates all of the theorems and functions declared in the class bodies of the abstract types ($CommMonoid$ and its supertypes such as $Monoid$). For example (2) instantiates to:

$$\forall x : Nat \cdot add(x, zero) = zero \Leftrightarrow x = zero \tag{4}$$

This in principal needs no proof, instead the proof is done on (1) so it is true for all $Monoid$s and the **Instance** statement requires it to be proved that add and $zero$ form a $CommMonoid$ (which requires them to be a $Monoid$).

Instantiating the theorems in this manner makes future proofs considerably easier as the person doing the proofs does not have to manually instantiate the theorems whilst proving. When developing theorems in B$^\sharp$ it was found that the theorems were considerably more concise than the equivalent theorems developed in the Event-B case study [9]. A large part of this was due to the instantiation mechanism, however, other features of the B$^\sharp$ language helped, including the B$^\sharp$ type system, and predicates not being a separate syntactic category in B$^\sharp$ (unlike Event-B). It was also notable that due to the IDE features built into the B$^\sharp$ tool such as syntax aware autocompletion and on the fly error highlighting, it was easier and faster in our opinion to develop theories than when using the Theory Plug-in tools.

3 Discussion and Future Work

B$^\sharp$ bears similarities to HOL [7] style languages, with a type class style feature [12]. Languages such as Coq [3] and Isabelle/HOL [8] with similar features have been used to construct large libraries of mathematical types. These features also have a lot in common with Algebraic specification language [6], where parameterised programming [11] allows generic types to be constricted to type specifications. The unique feature of this work is the translation to the set theoretic syntax used by Event-B, and the interaction with its tools.

Developing mathematical theorems in B$^\sharp$ highlighted problems and improvements that could be made to the B$^\sharp$ language. For example **Instance** statements infer their supertype from their parametric context, which is the wrong way round; supertypes, if required, should be explicit. In **Class** statements what constitutes a supertype needs to be extended. An attempt was made to create a class of all commutative monoids on the naturals with equality as the equivalence operator, whilst this was possible it would have been easier if **Instance**s could be supertypes. These issues will be addressed with updates to the B$^\sharp$ syntax.

The B$^\sharp$ tool does not interact directly with the Rodin interactive prover (this is done through the translation to existing Event-B and the current tools). The B$^\sharp$ type system is not the same as the Event-B type system, as it allows subtyping (and requires functions to explicitly state their return types). Prover rules and tactics could be generated for the interactive prover to allow this additional information to be used, in some cases this would allow proofs to be discharged automatically, further reducing the proof burden on the user.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.

[2] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transfer*, 12(6):447–466, 2010.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*, *Texts in Theoretical Computer Science: An EATCS Series.* Springer, 2004.

[4] Chris Bogdiukiewicz, Michael J. Butler, Thai Son Hoang, Martin Paxton, James Snook, Xanthippe Waldron, and Toby Wilkinson. Formal development of policing functions for intelligent systems. In *Proc. of 28th Int. Symp. on Software Reliability Engineering, ISSRE 2017*, pages 194–204. IEEE, 2017.

[5] Michael J. Butler and Issam Maamria. Practical theory extension in Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013.

[6] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10(1):27–52, 1978.

[7] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[8] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[9] James Snook. Technical note: building abstract mathematical types in Event-B. Working paper, University of Southampton, April 2019.

[10] James Snook, Michael J. Butler, and Thai Son Hoang. Developing a new language to construct algebraic hierarchies for Event-B. In Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, editors, *Proc. of 4th Int. Symp. on Dependable Software Engineering: Theories, Tools, and Applications, SETTA 2018*, volume 10998 of *Lecture Notes in Computer Science*, pages 135–141. Springer, 2018.

[11] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Data type specification: Parameterization and the power of specification techniques. In *Proc. of 10th Ann. ACM Symp. on Theory of Computing, STOC '78*, pages 119–132. ACM, 1978.

[12] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. of 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '89*, pages 60–76. ACM, 1989.

A Formal Framework for Consent Management

Shukun Tokas and Olaf Owe

Department of Informatics, University of Oslo, Norway
{shukunt,olaf}@ifi.uio.no

Introduction

In response to the emerging privacy concerns, the European Union (EU) has approved the General Data Protection Regulation (GDPR) [1] to strengthen and impose data protection rules across the EU. This regulation requires controllers that process personal data of individuals within EU and EEA, to process personal information in a ”lawful, fair, and transparent manner”. Article 6 and Article 9 of the regulation [1] provide the criteria for lawful processing, such as consent, fulfillment of contractual obligation, compliance with a legal obligation etc. The regulation treats consent as one of the guiding principles for legitimate processing, and Article 7 [1] sets out the conditions for the processing personal data (when relying on consent).

Consent is defined as “any freely given, specific, informed and unambiguous indication of the data subject’s wishes by which he or she, by a statement or by a clear affirmative action, signifies agreement to the processing of personal data relating to him or her” [1]. In particular, a data subject‘s consent reflects her choices/agreements in terms of the processing of personal data. These privacy requirements can be expressed through privacy policies, which are used to regulate the processing of personal data. The privacy requirements in the GDPR (as well as other privacy regulations) are defined informally, therefore, to avoid ambiguity the policy language equipped with a formal semantics is essential [2, 3]. We have previously studied static aspects of privacy policies and policy compliance from a formal point of view, a brief overview is given in [4]. Here, we look at a formal approach to address consent at the modeling level.

The aim of this work is to design a formal framework for consent management where a data subject can change her privacy settings through predefined interface definitions, which could be seen as part of a library system. The data subjects are seen as system users without knowledge of the underlying program. The framework consists of predefined language constructs for specifying privacy policies and consent, compliance and consent checking, and a semantics. We prove a notion of compliance regarding consent. To make a general solution, we consider a high-level modeling language for distributed service-oriented systems, building on the paradigm of *active objects* [5, 6] .

Central to the design of this framework are *(a)* a policy definition language that allows specification of privacy policies; *(b)* a formalization of policy compliance; *(c)* integration of privacy policies in a programming language; *(d)* a run-time system for dynamic checking of privacy compliance, with built-in consent management. The framework covers essential GDPR aspects, providing practical means to support for *privacy by design* (Article 25, Recital 78 [1]) and *data subject access request* (Article 7, Recital 63 [1]). It is essential that the policy terminology establishes precise link between the law and the program artifacts. For this, we let privacy policies and consent definitions be expressed in terms of several predefined names, reflecting standard terminology (names can be added as needed). Since the data subject is not always a legal scholar or program developer, it is necessary that the policy terminology used towards the data subject is simple but with a formal connection to the underlying programming elements. The rest of this abstract will provide motivation and basic notions related to privacy and consent specification, but we will omit details for the runtime system.

Language Setting

In the setting of active objects, the objects are autonomous and execute in parallel, communicating by so-called asynchronous method invocations. An Object-local data structure is defined by data types. We assume interface abstraction, i.e., remote field access is illegal and an object can only be accessed though an interface. This allows us to focus on major challenges of modern architectures, without the complications of low-level language constructs related to the shared-variable concurrency model. The programs we consider are defined by a sequence of declarations of interfaces (containing method declaration), classes (containing class parameters, fields, methods and an initialization) and data type definitions. Classes are defined by an imperative language and data types and associated functions by a functional language.

Interfaces may have a number of superinterfaces, letting the predefined interface *Any* be the most general interface (supported by any object). We let interface *Sensitive* be a subinterface of *Any*, corresponding to a system component (active object) with personal data. By static checking it is ensured that any object receiving personal data must support the interface *Sensitive* [4]. And we let interface *Principal* be a subinterface of *Any* corresponding to a system user, be it a person, an organization, or other identifiable actor. Interface *Subject* is a subinterface of both *Principal* and *Sensitive*, and corresponds to what GDPR refers to as "data subjects". Interface *Sensitive* defines methods for accessing and resetting consented (and default) policies, by the data subject. Interface *Subject* offers methods for consent management including functionality for requesting and updating consent settings.

Runtime aspects

At runtime there will be a number of concurrent objects containing data values (of some type), and communicating by method calls. Data values with personal information will be tagged. The tags reflect associated consent information. This meta information is not directly accessible to the programmer or system user, but is understood by the runtime system, to restrict access to private data. Our framework includes a general solution for subjects to observe and change their privacy settings, and a way to delete private data (soft delete). The tags include privacy information such as identification of the *subject*, as well as *role*, *purpose* and *access rights*, specifying who (the roles that can use the data), why (purpose for which the data can be used), and how (kind of operation or access allowed on the data).

Formalization of Consent and Privacy Policies

We let privacy and consent definitions be expressed in terms of *role*, *purpose* and *access rights* for a given subject, where each of these range over a set of names, including predefined names reflecting standard terminology, names can be added as needed. A *role* is given by a name such as *Doctor*, *Nurse*, *Patient*, also arranged in a directed acyclic graph with a (transitive and anti-symmetric) less-than relation $<$. At the programming level, roles are reflected by interfaces or *Principals.* A Principal object may implement multiple interfaces to support several roles. A *purpose* is given by a name such as *health_care, advertising, treatment, billing, research.* The *purpose* names are arranged in a directed acyclic graph with a (transitive and anti-symmetric) less-than relation $<$, for instance $treatment < health\_care$ or $research < health\_care$. For *access-rights* we consider a fixed terminology for describing access rights, with *read*, *incr*, *self* and *write* access to the data. Access rights are given by a complete lattice, with $\sqcup$ and $\sqcap$ as lattice operators, with *full* (for full access) as top element and *no* (for no access) as bottom element. Furthermore, *read* gives read access, *write* gives over-write access, *incr* gives incremental access (adding an element to a list or set without reading the other

A	$::=$	$read \mid incr \mid write \mid self$	basic access rights
		$\mid no \mid full \mid rincr \mid wincr$	abbreviated access rights
		$\mid A \sqcap A \mid A \sqcup A$	combined access rights
$\mathcal{P}$	$::=$	(I, R, A)	policies
$\mathcal{P}s$	$::=$	$\{\mathcal{P}^*\} \mid \mathcal{P}s \sqcap \mathcal{P}s \mid \mathcal{P}s \sqcup \mathcal{P}s$	policy sets
$\mathcal{CP}$	$::=$	$[S, \mathcal{P}s] \mid [I, \mathcal{P}s]$	basic consent declarations
$\mathcal{CP}s$	$::=$	$\{\mathcal{CP}^*\} \mid \mathcal{CP}s \sqcap \mathcal{CP}s \mid \mathcal{CP}s \sqcup \mathcal{CP}s$	sets of consent declarations

Figure 1: BNF syntax definition of the policy language. I ranges over interface names, R over purpose names, and S over data subjects. Superscript $*$ denotes repetition.

purpose $treatm, health\_care$
  where $treatm < health\_care$
policy $\mathcal{P}_{MyDoc} = (''dr.Hansen'', treatm, full)$ `// specific policy`
policy $\mathcal{P}_{Doc} = (Doctor, treatm, rincr)$ `// general policy`
policy $\mathcal{P}_{Nurse} = (Nurse, treatm, read)$ `// general policy`
consent $\mathcal{CP}_{my} = [''Olaf'', \{\mathcal{P}_{MyDoc}, \mathcal{P}_{MyDoc}, \mathcal{P}_{Nurse}\}]$ `//consent specification`

Figure 2: Sample purpose, policy, and consent definitions. Subjects are identified by strings.

elements), and *self* gives the subject (full) access to data about itself. Thus $read \sqcup incr$ gives a principal read access and incremental access, but not general write access. This is quite useful in many connections and therefore we introduce *rincr*, as an abbreviation for it.

The language syntax for defining access rights, privacy policies, and consent is summarized in Figure 1 and some sample policies are found in Figure 2. A privacy policy $\mathcal{P}$ is given by a who-why-how triple (I, R, A), where I, R and A range over interfaces, purposes and access rights, respectively, each with their own hierarchy, following [4]. Policy sets form a lattice with $\sqcup$ and $\sqcap$ as lattice operators and with the empty set as the bottom element. A consent specification on data is given by the *subject* identity and a set of who-why-how policies and have the form $\{Subject, \{\mathcal{P}^*\}\}$. For example, we may specify consent for a patient p by $\{p, \{(Doctor, treatm, rincr), (Nurse, treatm, read)\}\}$. Private data in general is tagged with a single consent declaration, assuming the data concern the privacy of a single data subject.

References

[1] European Commission. The General Data Protection Regulation (GDPR) regulations of the European Parliament and of the Council. Accessed 28 Apr. 2019.

[2] R. Pardo and D. L. Métayer. Analysis of privacy policies to enhance informed consent. arXiv preprint arXiv:1903.06068, 2019.

[3] D. Le Métayer. Formal methods as a link between software code and legal rules. In G. Barthe, A. Pardo, G. Schneider, editors, *Proc. of 9th Int. Conf. on Software Engineering and Formal Methods, SEFM 2011*, volume 7041 of *Lecture Notes in Computer Science*, pp. 3–18. Springer, 2011.

[4] S. Tokas, O. Owe, and T. Ramezanifarkhani. Language-based mechanisms for privacy by design. In *Revised Selected Papers from 14th IFIP Int. Summer School on Privacy and Identity Management, IFIP Advances in Inform. and Commun. Techn.* Springer, to appear.

[5] O. Nierstrasz. A tour of Hybrid—a language for programming with active objects. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pp. 67–182. Prentice-Hall, 1992.

[6] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Softw. Syst. Modeling*, 6(1):39–58, 2007.

Statically Derived Data Access Patterns for NUMA Architectures [*]

Gianluca Turin, Einar Broch Johnsen, and S. Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{gianlutu,einarj,sltarifa}@ifi.uio.no

Abstract

A parallel program can be thought of as a collection of communicating tasks. When executing such a parallel program in a distributed setting, remote data access may introduce significant communication latency. Network contention is the main factor affecting latency. Therefore, it is important to reduce the amount of remote data access, in terms of distance travelled. Network contention can be significantly affected by the long distances covered while accessing remote data in architectures such as large mesh-network or torus-network machines. For example, BlueGene/P is a huge torus topology which may have as many as 130k processing elements. In particular, it is important to reduce data movement on NUMA machines based on such topologies. Several techniques have been developed for this purpose; for example, there are different heuristics to efficiently schedule tasks given their dependency graph.

In this work, we consider a static analysis of data locality in programs to reduce latency of hardware communication by means of topology-aware task scheduling, and thus to improve execution performance on large high performance computing systems. Our analysis enables the automatic extraction of dependency graphs, instead of manually specifying the communication pattern of an application or running it in a traced execution to collect this information. Our analysis is formalised as a type and effects system, we here describe the basic idea without explaining the details of the formalisation.

1 Motivation

The main resources in a large parallel machine are its cores and the network infrastructure. Optimal resources usage is key to tighten the gap between theoretical peak performance and actual peak performance. For example, IBM BlueGene/P [6] features flat 3D torus networks where each node connects directly to its six neighbours. Unless each node only communicates with its physically nearest neighbours, nodes have to share network links with other communication. Unless the physical placement of application processes matches the communication characteristics of the application, such sharing can result in significant communication contention and performance loss [3].

In theory any mapping of processes to cores is possible, in practice different systems have different restrictions on what mappings they allow. For example, IBM Blue Gene/P allows different combinations of process mappings along the X, Y, Z dimensions of the 3D torus, and BG/P systems consist of four cores per compute node, which can be considered to be a fourth dimension represented as T. An application can be mapped using different mappings, such as $TXYZ$, $XYZT$, and $TYXZ$. Other mappings that do not form a symmetric ordering of ranks in one of these orders are not supported. Thus, the "best performance" we can achieve is artificially restricted by this requirement [3].

[*] *ADAPt: Exploiting Abstract Data-Access Patterns for Better Data Locality in Parallel Processing* (www.mn.uio.no/ifi/english/research/projects/adapt/).

In most supercomputers, tasks are scheduled statically but they can migrate from the queue of one core to another due to job stealing or automatic workload balancing strategies. These strategies generally improve performance, but they may increase distances and total latency during execution.

2 A static analysis for data access patterns

To formalise information about locality in a fine-grained way on NUMA architectures, we propose a static analysis expressed using types and effects (e.g., [2]), which combines types with *locations* for a small core language targeting parallel hardware, where each assignment, function call or expression evaluation may involve long distance remote data access. In this language, new locations can be dynamically created and new tasks dynamically spawned. A task scheduling strategy decides on which core the tasks will execute. Our notion of locality is inspired by the Partitioned Global Address Space model (PGAS) [8]. This programming model provides a memory abstraction on parallel hardware that considers the memory partitioned in places: each task can only access its local memory directly and a global shared space is used across the computing nodes to share data among tasks.

Since the distances covered by remote accesses can affect the performance significantly on large networks [4], our goal is to investigate whether making information about the data access patterns of the different tasks available to the task scheduler can improve the overall performance. For this purpose, information of the network topology needs to be provided to the static analyser. A formal description of a given network with a metric capturing distances, can be obtained by letting the metric between each pair of processors reflect, e.g., the number of hops, the average or the expected latency of a packet. A metric based on hop-bytes (hops per byte) can approximate the overall contention on the network [1, 5]. Although the metric does not capture hot-spots created on specific links, it is easily derivable and correlates well with the actual application performance when communication to computation ratio is high [4].

Consider as a simple example an assignment instruction $x = y$, where x and y are at different locations in memory. These locations can be captured in a type system such that x has a location type of the form $L_a \cdot T$ where the location L_a abstractly describes the memory location that must be accessed to write the variable x on the left hand side of the assignment, and T describes the (standard) type of the value that can be written. Similarly for y the type may look like $L_b \cdot T'$. Again, the element L_b indicates a location in memory that we need to read to obtain the value of y and the T' indicates the type of that value.

Example: computing the average of distributed data

Let us consider a task for computing the average of a large distributed array: a set of processes have to locally reduce their array partition to the sum of all the elements. Then all those values have to be sent to a master process that will compute and print the total average.

Many parallel languages and libraries would be suitable to code this specification on a NUMA machine, by relying on message passing to achieve synchronisation. Nevertheless the recent trend has been to develop languages and libraries for NUMA machines providing the same programmability of standard programming languages [7].

In our core language such code could be implemented as in Figure 1. In this example it is possible to estimate the cost of the assignment of the variables in the array p_sums by checking the locality information coded in their location types.

```
task<$L_{\mathcal{T}}$> worker(sums:L·$L_{\mathcal{T}} \cdot \mathbb{N}$){
    local $L_{\text{p\_sum}}$, L, $L_i$, $L_{a_1}$, ..., $L_{a_{25}}$;
    var id: L·ℕ;
    var p_sum:$L_{\text{p\_sum}} \cdot \mathbb{N}$;
    var a[25]: $L_{a_1 \ldots a_{25}} \cdot \mathbb{N}$;
    var i: $L_i \cdot \mathbb{N}$;

    for(i=0; i<25; i++) { a[i] = random(); p_sum = p_sum + a[i]; };

    *sums = p_sum; }
{
    local $L_i$, $L_{\text{p\_sums}_1}$, ..., $L_{\text{p\_sums}_4}$, $L_{\text{avg}}$;
    var p_sums[4]: $L_{\text{p\_sums}_1 \ldots \text{p\_sums}_4} \cdot \mathbb{N}$; // initially null
    var avg: $L_{\text{avg}} \cdot \mathbb{N}$; // initially null
    var i: $L_i \cdot \mathbb{N}$;

    for(i=1; i<5; i++) { spawn <$L_{\text{p\_sums}_i}$>worker(&p_sums[i]); };

    for(i=1; i<5; i++) { while(p_sums[i]!=null); }; // active wait for p_sums

    for(i=0; i<5; i++) { avg = avg + p_sums[i]; };
    avg = avg / 100;
    print avg; }
```

Figure 1: Example: location types for a task computing the average of a large distributed array.

Given a sequence of five contiguous cores, the optimal scheduling is to place the master process, calculating the average, in the middle. Such configuration will give also the minimal total cost in our analysis.

While the distance covered by four integer values is far away to affect performances, a common scenario in parallel programming can be having streams of data converging to a filtering process. If few streams are enough to saturate the capacity of the network, more streams would require another filtering process. A scheduling able to avoid intersection of streams in this case would reduce network contention and thus latency.

Discussion and future work

When analysing a statement, we can use location types to estimate the cost of accesing data at different locations, according to the metric of the topology. This fine level of granularity assumes that a statement can remotely read and write variables, such that dependencies are evaluated on an instruction basis. The analysis then collects the derived costs according to the control flow; e.g., accumulating the cost of the two statements for sequential composition and taking the worst case for conditionals, and iterating over a for-loop. The granularity of the analysis and thereby the complexity of leveraging the analysis results, can be reduced as necessary by clustering locations to obtain a simpler dependency graph from the more complex instructions dependency graph. This way, the derived data access patterns can be simplified enough to be exploited by a scheduling strategy with acceptable overhead.

Since the analyser knows the topology metric, it can calculate for each task the possible cost of executing that task on the different cores. At runtime, the task scheduler can combine these statically derived data access patterns with runtime information in order to decide on the placement of a task. Our work so far is thus focused on how to statically derive data access patterns, in future work we plan to investigate how to extract these patterns automatically and to explore how they can be efficiently combined with runtime information by a task scheduling strategy, to improve data movement overhead on huge NUMA machines.

References

[1] T. Agarwal, A. Sharma, A. Laxmikant, and L. V. Kale. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proc. of 20th Int. Parallel Distributed Processing Symposium, IPDPS 2006*. IEEE, 2006.

[2] T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[3] P. Balaji, R. Gupta, A. Vishnu, and P. Beckman. Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems. *Comput. Sci. Res. Dev.*, 26(3–4):247–256, 2011.

[4] A. Bhatele. *Automating Topology Aware Mapping for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, 2010.

[5] A. Bhatele and L. V. Kale. Heuristic-based techniques for mapping irregular communication graphs to mesh topologies. In *Proc. of 13th IEEE Int. Conf. on High Performance Computing and Communications, HPCC 2011*, pages 765–771. IEEE, 2011.

[6] Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52(1.2):199–220, 2008.

[7] J. Kepner. HPC productivity: An overarching view. *Int. J. High Performance Comput. Appl.*, 18(4):393–397, 2004.

[8] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu. The asynchronous partitioned global address space model. In *Proc. of 1st ACM SIGPLAN Workshop on Advances in Message Passing*. ACM, 2010.

A Framework for Exogenous Stateless Model Checking

Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte

Department of Informatics, University of Oslo, Norway
{larstvei,einarj,rudi}@ifi.uio.no

1 Introduction

Different execution paths for sequential, deterministic programs are typically explored by means of unit tests, following standard industrial practice. In contrast, concurrent and distributed programs exhibit nondeterminism due to, e.g., message delays, message re-ordering and race conditions. To explore the different execution paths of such programs, one needs to solve issues of *controllability*, to guide the program execution along a desired path, and *state-space explosion*, since the number of possible executions can be very large.

Stateless model checking is a technique to explore the execution paths of a concurrent program [5]. While highly concurrent programs have a large number of execution paths, making full path exploration infeasible, many operations in a program are completely independent and can be reordered, resulting in paths that are equivalent. Only exploring one path per equivalence class yields a significant reduction in the number of executions that need to be explored. Partial order reduction [2] is a general technique that relates dependent events, based on a *happens-before* relation, and considers paths with the same *happens-before* as equivalent and thereby as members of the same *Mazurkiewicz trace* [7]. *Dynamic partial order reduction* [4] is an algorithm for stateless model checking guaranteed to execute at least one path per Mazurkiewicz trace. A variation of dynamic partial order reduction has been shown to achieve optimal reduction [1]. However, dynamic partial order reduction uses backtracking, which is challenging to parallelize. In contrast, *Offline stateless model checking*, developed by Huang [6], based on maximal causality reduction for shared memory systems, allows parallel path exploration and only requires forward execution.

We present a general framework for offline stateless model checking with partial order reduction, which allows parallel path exploration. Our work abstracts from a specific programming language by capturing the language semantics and path equivalence in terms of execution traces.

2 Traces and Relations over Events

We consider a programming language where the execution of a program can be abstractly captured as a trace τ, meaning a sequence of events $e_1 \cdots e_n$ from a (possibly infinite) set of events $\mathcal{E}$, and assume an associated runtime environment (or simulator) that can both emit such a trace and deterministically reproduce an execution that follows a given trace. If a trace τ represents only a prefix of a complete execution, we assume that after following τ the runtime continues non-deterministically until termination, producing a new trace.

Given some τ emitted from the runtime of a program, we are interested in the different *seed traces* that lead the execution down a path distinctly different from the one represented by τ. These seed traces can be computed from three relations over the set of events $\mathcal{E}$:

$e_i \overset{MHB}{\Longrightarrow} e_j$ if the event e_i *must* happen before e_j in all feasible executions

$e_i \oslash e_j$ if the order of e_i and e_j *may* affect the result of an execution.

$$e_i \overset{HB}{\Longrightarrow}_\tau e_j \qquad \text{if } e_i \text{ happened before } e_j \text{ in the trace } \tau.$$

For a programming language with a trace-based semantics, $\overset{MHB}{\Longrightarrow}$ can be stated at the level of the language semantics, $\oslash$ at the program level, and $\overset{HB}{\Longrightarrow}_\tau$ at the execution level. Intuitively, $\overset{MHB}{\Longrightarrow}$ ensures that the produced seed traces are possible according to the language semantics and $\oslash$ is used to establish equivalence between traces; e.g., a $\oslash$ relation that relates all events with each other will result in exploring all execution paths, while an empty $\oslash$ will result in only running a single execution. Lastly, the $\overset{HB}{\Longrightarrow}_\tau$ relation encodes the trace τ as a relation.

3 An Algorithm for Path Exploration

We present an algorithm for state exploration, where the execution of a program and the search for new execution paths are separated. Our algorithm is formulated as two procedures (see Algorithm 1 and 2); one that interacts with a runtime and keeps track of the state of the search, and one that generates new seed traces from a given explored trace.

Algorithm 1 Trace Exploration

```
procedure EXPLORE($Exec, P$)
    $E \leftarrow \emptyset$
    $Q \leftarrow \{\epsilon\}$
    while $\tau_s \in Q \setminus E$ do
        $\tau \leftarrow Exec(P, \tau_s)$
        $E \leftarrow E \cup PrefixesOf(\tau)$
        $Q \leftarrow Q \cup GenerateSeeds(\tau)$
    end while
    return $E$
end procedure
```

Algorithm 2 Seed Generation

```
procedure GENERATESEEDS($\tau$)
    $R \leftarrow \emptyset$
    for $\langle e_i, e_j \rangle \in \oslash_\tau$ do
        $\oslash'_\tau \leftarrow (\oslash_\tau \setminus \{\langle e_i, e_j \rangle\}) \cup \{\langle e_j, e_i \rangle\}$
        $\tau' \leftarrow$ Satisfy($\phi_{\overset{MHB}{\Longrightarrow}} \land \phi_{\oslash'_\tau}$)
        if $\tau' = \tau_s \cdot e_j \cdot \tau_r$ then
            $R \leftarrow R \cup \{\, \tau_s \cdot e_j \}$
        end if
    end for
    return $R$
end procedure
```

The search starts by exploring an arbitrary run of a program P, which corresponds executing the empty seed trace $Exec(P, \epsilon)$. From this trace, it generates new seed traces, that can be executed in any order. By keeping track of what executions have been explored (and their prefixes), we guarantee not to explore any execution twice. The search terminates when no seed trace is unexplored.

When generating new seed traces, we only consider pairs in $\oslash$. We define $\oslash_\tau = \overset{HB}{\Longrightarrow}_\tau \cap \oslash$, i.e. the interfering pairs in some particular trace τ, and call such pairs a *conflict*. For all conflicts, the algorithm attempts to reverse the order of these two events, while maintaining the order of all other conflicts. If we can find a trace, which respects the $\overset{MHB}{\Longrightarrow}$ relation and the reversed conflict, then we add it to the set of seed traces.

Concretely, we obtain the seed traces by encoding the relations as a SMT (Satisfiability modulo theories) problem, and solving each instance with Z3 [3]. Formulas are encoded as conjunctions of constraints $V(e_i) < V(e_j)$, where V is a mapping from events to integer variables. We only keep the prefixes of the traces generated by Z3 up to the reversed conflict, because we cannot generally make assumptions about what executions are possible after this point in the trace.

Several details are omitted in the presented algorithm, like measures to reduce regeneration of the same seed traces, minimizing the length of seed traces and comparing prefixes by their conflicts. An implementation of the full algorithm is available at https://github.com/larstvei/trace-exploration.

4 Example

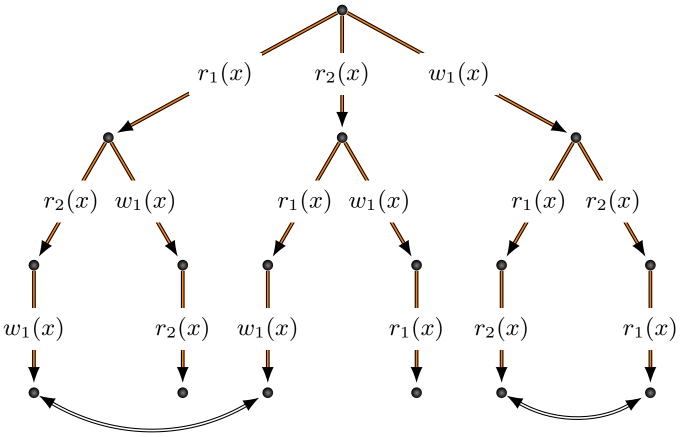

Figure 1: All executions of a simple read/write program.

Consider a simple shared memory language, where a process may atomically write a value to a variable, or read the current value of a variable. In this language, there is a direct correspondence between the statements of a program and the events of a trace. The $\overset{MHB}{\Longrightarrow}$ relation only need to ensure the thread-local ordering of events. If two events e_i, e_j operate on the same variable, and one of the operations is a write, then we say $e_i \oslash e_j$. The $\overset{HB}{\Longrightarrow}$ relation will relate an event e_i to e_j if e_i happened before e_j in the sequential trace.

Consider a simple program with three processes, where each process does one read or write operation, $r_1(x)$, $r_2(x)$ and $w_1(x)$ respectively. We show the possible executions of this program in Fig. 1, where the double-headed arrows indicate an equivalence with $\oslash$ relation as described above. The prototype implementation the yields four non-equivalent traces using this $\oslash$ relation; if we provide one where all events are in conflict, it outputs all the six execution paths.

5 Conclusion and Future Work

The presented framework for exogenous stateless model checking abstracts from any specific programming language and only considers relations over events, as described in Section 2. We showed how to instantiate the framework for a very simple read/write language. In future work, we plan to study the limitations of the framework in order to answer questions about optimality. For example, it is interesting to investigate a possible generalization to different notions of causality (e.g. can the framework be used for both partial order and maximal causality reduction [6]). Furthermore, we want to leverage the framework for a rich actor-based language and evaluate a fully parallelized implementation on complex programs.

References

[1] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proc. of 41st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '14*, pages 373–384. ACM, 2014.

[2] E. M. Clarke, O. Grumberg, M. Minea, and D. A. Peled. State space reduction using partial order techniques. *Int. J. Softw. Tools Technol. Transfer*, 2(3):279–287, 1999.

[3] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proc. of 14th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008*, volume 4963 of*Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '05*, pages 110–121. ACM, 2005.

[5] P. Godefroid. Model checking for programming languages using Verisoft. In *Conf. Record of 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '97*, pages 174–186. ACM, 1997.

[6] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proc. of 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '15*, pages 165–174. ACM, 2015.

[7] A. W. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Proc. of Advanced Course on Petri Nets 1986, Part II*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1987.

The Early π-Calculus in Ticked Cubical Type Theory

Niccolò Veltri[1] and Andrea Vezzosi[2]

[1] Dept. of Software Science, Tallinn University of Technology, Estonia
niccolo@cs.ioc.ee
[2] IT University of Copenhagen, Denmark
avez@itu.dk

The Nakano modality $\triangleright$ [10] is an operator that, when added to a logic or a type system, encodes time at the level of formulae or types. The formula $\triangleright A$ stands for "A holds one time step in the future", similarly the inhabitants of type $\triangleright A$ are the inhabitants of A in the next time step. The Nakano modality comes with a guarded fixpoint combinator $\mathsf{fix}_A : (\triangleright A \to A) \to A$ ensuring the existence of a solution for all guarded recursive equations in any type. Logically, this corresponds to a version of Löb's axiom for the $\triangleright$ modality.

Guarded recursion has been added to Martin-Löf dependent Type Theory in two different ways: using delayed substitutions as in Guarded Dependent Type Theory (gDTT) [2] or using ticks as in Clocked Type Theory (CloTT) [1]. In these settings, the Nakano modality is employed for constructing guarded recursive types, i.e. recursive types in which the recursive variables are guarded by $\triangleright$. These are computed using the fixpoint combinator at type U, which is the universe of small types. For example, the guarded recursive type of infinite streams of natural numbers is obtained as $\mathsf{Str} = \mathsf{fix}_{\mathsf{U}} X. \, \mathbb{N} \times \triangleright X$ and it satisfies the type equivalence $\mathsf{Str} \simeq \mathbb{N} \times \triangleright \mathsf{Str}$. Recursively defined terms of guarded recursive types are causal and productive by construction.

Dependent type theories with guarded recursion have proved themselves suitable for the development of denotational semantics of programming languages, as demonstrated by Paviotti et al's formalization of PCF [11] and Møgelberg and Paviotti's formalization of FPC in gDTT [8]. Here we continue on this line of work by constructing a denotational model of Milner's early π-calculus in a suitable extension of CloTT. Traditionally, the denotational semantics of π-calculus is developed in specific categories of (presheaves over) profunctors [3] or domains [12, 6]. Fundamentally, the semantic domains have to be sufficiently expressive to handle the non-deterministic nature of π-calculus processes. In domain theoretic semantics, for example, this is achieved by employing powerdomains. Synthetic analogues of these constructions are not available in guarded type theories such as gDTT or CloTT, but it can be constructed if we set our development in extensions of these type systems with Higher Inductive Types (HITs), a characterizing feature of Homotopy Type Theory (HoTT).

We work in Ticked Cubical Type Theory (TCTT) [9], an extension of Cubical Type Theory (CTT) [5] with guarded recursion and the ticks from CloTT. CTT is an implementation of a variant of HoTT, giving computational interpretation to its characteristic features: the univalence axiom and HITs. Roughly speaking, the univalence axiom provides an extensionality principle for types, allowing to consider equivalent types as equal. A HIT A can be thought of as an inductive type in which the introduction rules not only specify the generators of A, but can also specify the generators of the higher equality types of A. The latter are commonly referred to as *path constructors*, due to the interpretation of equality proofs as paths in HoTT. For example, here is a presentation of the countable powerset datatype as a HIT [4], that will serve as our synthetic powerdomain:

$$\frac{}{\varnothing : \mathsf{P}_\infty\, A} \qquad \frac{a : A}{\{a\} : \mathsf{P}_\infty\, A} \qquad \frac{s : \mathbb{N} \to \mathsf{P}_\infty\, A}{\bigcup s : \mathsf{P}_\infty\, A}$$

$$\frac{x, y : \mathsf{P}_\infty\, A}{\_ : x \cup y = y \cup x} \qquad \frac{x, y, z : \mathsf{P}_\infty\, A}{\_ : (x \cup y) \cup z = x \cup (y \cup z)} \qquad \frac{x : \mathsf{P}_\infty\, A}{\_ : x \cup \varnothing = x} \qquad \frac{x : \mathsf{P}_\infty\, A}{\_ : x \cup x = x}$$

$$\frac{s : \mathbb{N} \to \mathsf{P}_\infty\, A \quad n : \mathbb{N}}{\_ : s\, n \cup \bigcup s = \bigcup s} \qquad \frac{s : \mathbb{N} \to \mathsf{P}_\infty\, A \quad x : \mathsf{P}_\infty\, A}{\_ : \bigcup s \cup x = \bigcup(\lambda n. s\, n \cup x)} \qquad \text{the 0-truncation constructor}$$

The type $\mathsf{P}_\infty\, A$ has three generators: the empty subset, the singleton constructor and countable union. Binary union $x \cup y$ is defined as the countable union of the sequence $x, y, y, y, \ldots$ The path constructors are the equations of the theory of countable-join semilattices, while the 0-truncation constructor forces $\mathsf{P}_\infty\, A$ to be a "set" in the sense of HoTT, that is a type with trivial higher paths. A membership relation $\in : A \to \mathsf{P}_\infty\, A \to \mathsf{U}$ is definable by induction on the second argument.

TCTT also has ticks. The Nakano modality is now indexed over the sort of ticks, $\triangleright(\alpha : \mathsf{tick}).A$, and its inhabitants are to be thought of as dependent functions taking in input a tick β and returning an inhabitant of $A[\beta/\alpha]$. So ticks correspond to resources witnessing the passing of time that can be used to access values available only at future times. We write $\triangleright A$ for $\triangleright(\alpha : \mathsf{tick}).A$ when α does not occur free in A. The $\triangleright$ modality is an applicative functor, its unit is called next. Ticks allow to extend the applicative structure to dependent types.

For the specification of the π-calculus syntax, we assume the existence of a countable set of names. Practically, for every natural number n, we assume given a type $\mathsf{Name}\, n$, the set containing the first n names. Each process can perform an output, an input or a silent action. The type of actions is indexed by two natural numbers, representing the number of free names and the sum of free and bound names, respectively. The input action binds the input name.

$$\frac{ch, v : \mathsf{Name}\, n}{\mathsf{out}\, ch\, v : \mathsf{Act}\, n\, n} \qquad \frac{ch : \mathsf{Name}\, n}{\mathsf{inp}\, ch : \mathsf{Act}\, n\, (n+1)} \qquad \frac{}{\tau : \mathsf{Act}\, n\, n}$$

The π-calculus syntax includes the nil process, prefixing, binary sums, parallel composition, restriction, a matching operator and replication.

$$\frac{}{\mathsf{end} : \mathsf{Pi}\, n} \qquad \frac{a : \mathsf{Act}\, n\, m \quad P : \mathsf{Pi}\, m}{a \cdot P : \mathsf{Pi}\, n} \qquad \frac{P : \mathsf{Pi}\, n \quad Q : \mathsf{Pi}\, n}{P \oplus Q : \mathsf{Pi}\, n} \qquad \frac{P : \mathsf{Pi}\, n \quad Q : \mathsf{Pi}\, n}{P \| Q : \mathsf{Pi}\, n}$$

$$\frac{P : \mathsf{Pi}\, (n+1)}{\nu P : \mathsf{Pi}\, n} \qquad \frac{x, y : \mathsf{Name}\, n \quad P : \mathsf{Pi}\, n}{\mathsf{guard}\, x\, y\, P : \mathsf{Pi}\, n} \qquad \frac{P : \mathsf{Pi}\, n}{!P : \mathsf{Pi}\, n}$$

The processes in $\mathsf{Pi}\, n$ are quotiented by a structural congruence relation $\approx$, which, among other things, characterizes the replication operator in terms of parallel composition: given a process $P : \mathsf{Pi}\, n$, we have $!P \approx P \| !P$. The early operational semantics is inductively defined as a type family $- [-] \mapsto - : \mathsf{Pi}\, n \to \mathsf{Label}\, n\, m \to \mathsf{Pi}\, m \to \mathsf{U}$. Following [7], the type $\mathsf{Label}\, n\, m$ of transition labels include a silent action, free and bound outputs, and free and bound inputs.

For the denotational semantic domain, we consider the guarded recursive type

$$\mathsf{Proc} := \mathsf{fix}_{\mathbb{N} \to \mathsf{U}} X.\, \lambda n.\, \mathsf{P}_\infty(\mathsf{Step}\, (\lambda m. \triangleright \alpha. X\, \alpha\, m)\, n)$$

where $\mathsf{Step}\, Y\, n := \Sigma(m : \mathbb{N}).\, \mathsf{Label}\, n\, m \times Y\, m$. In other words, $\mathsf{Proc}\, n$ is the type satisfying the type equivalence $\mathsf{Proc}\, n \simeq \mathsf{P}_\infty(\Sigma m : \mathbb{N}.\, \mathsf{Label}\, n\, m \times \triangleright \mathsf{Proc}\, m)$. Let Unfold be the right-to-left morphism underlying the latter equivalence. To each syntactic process $P : \mathsf{Pi}\, n$ we associate a semantic process $[\![P]\!] : \mathsf{Proc}\, n$. The interpretation respects the structural congruence relation, that is $P \approx Q$ implies $[\![P]\!] = [\![Q]\!]$. The early operational semantics transitions are modelled using the membership operation: given $P\,[a]{\mapsto}Q$ with $a : \mathsf{Label}\,n\,m$, we have $(m, a, \mathsf{next}\,[\![Q]\!]) \in \mathsf{Unfold}\,[\![P]\!]$. Nevertheless, Proc is not closed under name substitutions. Therefore, to obtain a sound interpretation of π-calculus, we need to move to the following domain:

$$\begin{aligned}\mathsf{PiMod}\,n := \Sigma(P : \Pi(m : \mathbb{N}).(\mathsf{Name}\,n \to \mathsf{Name}\,m) \to \mathsf{Proc}\,m).&\\ \Pi(m, m' : \mathbb{N})(f : \mathsf{Name}\,m \to_{\mathsf{inj}} \mathsf{Name}\,m')(\rho : \mathsf{Name}\,n \to \mathsf{Name}\,m).&\\ \mathsf{mapProc}\,f\,(P\,m\,\rho) = P\,m'\,(f \circ \rho)&\end{aligned}$$

where $A \to_{\mathsf{inj}} B$ is the type of injective maps between A and B, while $\mathsf{mapProc}$ corresponds to the action of the functor Proc on injective renamings.

TCTT provides an extensionality principle for guarded recursive types: strong bisimilarity is equivalent to path equality [9]. For $\mathsf{Proc}\,n$, this says that semantic early bisimilarity is equivalent to path equality. In our work, we also define a syntactic notion of early bisimilarity and early congruence and we prove the denotational semantics fully abstract wrt. it.

We formalized the whole development in our own version of the Agda proof assistant based on TCTT, called Guarded Cubical Agda, an extension of Vezzosi et al's Cubical Agda [13].

Acknowledgments Niccolò Veltri was supported by the ESF funded Estonian IT Academy research measure (project 2014-2020.4.05.19-0001).

References

[1] P. Bahr, H. B. Grathwohl, and R. E. Møgelberg. The clocks are ticking: No more delays! In *Proc. of 32nd Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS 2017*, pages 1–12. IEEE, 2017.

[2] A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In B. Jacobs and C. Löding, editors, *Proc. of 19th Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2016*, volume 9634 of *Lecture Notes in Computer Science*, pages 20–35. Springer, 2016.

[3] G. L. Cattani, I. Stark, and G. Winskel. Presheaf models for the pi-calculus. In E. Moggi and G. Rosolini, editors *Proc. of 7th Int. Conf. on Category Theory and Computer Science, CTCS 1997*, volume 1920 of *Lecture Notes in Computer Science*, pages 106–126. Springer, 1997.

[4] J. Chapman, T. Uustalu, and N. Veltri. Quotienting the delay monad by weak bisimilarity. *Math. Struct. Comput. Sci.*, 29(1):67–92, 2019.

[5] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *IFCoLog J. Log. Appl.*, 4(10):3127–3170, 2017.

[6] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully-abstract model for the pi-calculus (extended abstract). In *Proc. of 11th Ann. IEEE Symp. on Logic in Computer Science, LICS 1996*, pages 43–54. IEEE, 1996.

[7] R. Milner, J. Parrow, and D. Walker. Modal logics for mobile processes. *Theor. Comput. Sci.*, 114(1):149–171, 1993.

[8] R. E. Møgelberg and M. Paviotti. Denotational semantics of recursive types in synthetic guarded domain theory. *Math. Struct. Comput. Sci.*, 29(3):465–510, 2019.

[9] R. E. Møgelberg and N. Veltri. Bisimulation as path type for guarded recursive types. *Proc. ACM Program. Lang.*, 3(POPL):4:1–4:29, 2019.

[10] H. Nakano. A modality for recursion. In *Proc. of 15th Ann. IEEE Symp. on Logic in Computer Science, LICS 2000*, pages 255–266. IEEE, 2000.

[11] M. Paviotti, R. E. Møgelberg, and L. Birkedal. A model of PCF in guarded type theory. *Electron. Notes Theor. Comput. Sci.*, 319:333–349, 2015.

[12] I. Stark. A fully abstract domain model for the pi-calculus. In *Proc. of 11th Ann. IEEE Symp. on Logic in Computer Science, LICS 1996*, pages 36–42. IEEE, 1996.

[13] A. Vezzosi, A. Mörtberg, and A. Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proc. ACM Program. Lang.*, 3(ICFP):87:1–87:29, 2019.

Approaches to Thread-Modular Static Analysis*

Vesal Vojdani, Kalmer Apinis, and Simmo Saan

Institute of Computer Science, University of Tartu, Estonia
{vesal,kalmera,ssaan}@ut.ee

1 Introduction

As multi-threaded operating systems are increasingly used in safety-critical settings, verification and analysis techniques have had to adapt. Different approaches to analyzing concurrent programs have been proposed, but due to differences in formalism, it is not clear how they relate. For now, we focus on (abstract) denotational semantics of concurrency via nested fixpoints [Ferrara, 2008]. We will reformulate the specific analysis of Miné [2012] in constraint-based form. We preserve the key ideas, but make some simplifications for the sake of clarity. In particular, we use the same abstract domain for the analysis as well as the communication between threads.

The use of explicit constraint system variables for thread communication makes the ideas more clear; in particular, the parallels with rely-guarantee reasoning is transparent in our formulation. There are, however, also practical advantages (as well as drawbacks) to constraint-based static analysis. Mainly, as more intermediate values are preserved, it is easier to proceed with incremental analysis. The drawback is increased memory consumption.

2 Concrete Thread-Modular Semantics

We consider programs as flow graphs: a set of nodes N and edges E of the form (u, s, v) where u and v are the program points in N and s is an elementary instruction, i.e., either an assignment or a conditional guard. There is also a dedicated elementary edge that initializes the program. These instructions manipulate sets of program states, each state being an environment mapping from variables to integers; this set we denote with $\rho \in D = \mathcal{P}(V \to \mathbb{N})$. We assume the semantics of an instruction s is given by $[\![s]\!] \colon D \to D$. We describe the set of states ρ_u that reach a program point u as the least solution to the following constraint system:

$$\rho_u \supseteq [\![s]\!]\rho_v \qquad\qquad (u, s, v) \in E$$

In the concurrent setting, we assume we are given a set of threads $\mathcal{T}$ and each $t \in \mathcal{T}$ has its own flow graph (N_t, E_t). The concrete semantics of a (sequentially consistent) concurrent program can be given as the non-deterministic execution of all thread interleavings, but here we immediately give a more thread-modular, yet concrete, view of interleaving semantics suggested in a subsequent paper by Miné [2014]. We encode the control state within the mappings ρ by having dedicated program counter variables pc_t, so that $\rho(\mathrm{pc}_t)$ returns the (unique ID in $\mathbb{N}$ associated with) the program point in N_t that thread t will execute next. We can then apply $[\![s]\!]$ to these extended states, leaving the program counters untouched. The essence of the thread-modular view is that for each thread t, the effect of its execution is given by a function $I_t \colon D \to D$. With this function, the constraint system for concurrent programs remains almost

*This work was supported by the Estonian Research Council grant PSG61.

unchanged:

$$\begin{aligned}
\rho_v &\supseteq [\![s]\!]\rho_u && (u,s,v) \in E_t && (1)\\
\rho_u &\supseteq \{\sigma \in I_{t'}(\rho_u) \mid \sigma(\mathrm{pc}_t) = u\} && u \in N_t,\ t' \neq t && (2)
\end{aligned}$$

In addition to local steps (1), we take into account the effect of other threads at each program point (2). The second constraint relies on the program counter variables to restrict influences. Given a complete description of the interference from other threads, the above system thus computes the interleaving semantics. Now, this interference can also be successively built by adding the following constraint:

$$I_t \supseteq \{(\sigma,\sigma') \mid \sigma \in \rho_u, \sigma' \in [\![s]\!]\rho_v\} \qquad (u,s,v) \in E_t \qquad (3)$$

The idea is to solve this in a nested fashion by first stabilizing the first two equations and then propagating the effect to other threads. This nesting is more evident in the denotational form [Miné, 2014], but the key point is that we can now understand different abstractions of concurrent communication by considering more abstract representations of the thread-interference space $D \to D$.

3 Interference Abstraction and Global Invariants

In this abstract, we will consider only the abstraction used in the original paper [Miné, 2012]. This is obtained by flow-insensitive and non-relational abstraction of communications. Thus, control variables are forgotten and thread effects are collapsed to variables mapping to their updated values, i.e., $V \to \mathcal{P}(\mathbb{N})$. Using this abstraction, we can write the constraints more informatively by using auxillary constraint variables R and G that highlight the correspondence to rely-guarantee reasoning:

$$\begin{aligned}
\rho_u &\sqsupseteq R_t && u \in N_t && (4)\\
G_t(x) &\sqsupseteq \rho_u(x) && (\_, s, u) \in E_t,\ x \in \mathsf{write}_s && (5)\\
\rho_v &\sqsupseteq [\![s]\!]\rho_u && (u,s,v) \in E_t && (6)\\
R_t &\sqsupseteq G_{t'} && t' \neq t && (7)
\end{aligned}$$

Each thread t can rely on what is guaranteed by all other threads, as expressed by the last constraint and enforced upon each program point by the first. The guarantee of a thread is computed by the second constraint, which only takes into account the values written at a given program point. This system is sound and no less precise than the formulation by Miné [2012], which applies the thread influences lazily; however, we can show that whenever expressions are evaluated, the results will coincide.

4 Scheduling Abstraction and Privatization

For priorities and mutexes, we follow the scheduler model of Miné [2012]. There are a fixed number of threads with unique priorities. The scheduler runs the thread with highest priority that is *ready*. We have lock, unlock, and yield instructions that can make threads not *ready*. We have previously proposed a constraint-based formulation based on privatization [Vojdani et al., 2016] for a simpler concurrency model. We track for each variable x, the set of mutexes always held whenever accessing x. This set is stored in $\Lambda(x)$. We further assume a sound must-lockset analysis is given that handles locks and unlocks, such that the set of locks definitely held at a given program point u is given by $\mathsf{locks}(\rho_u)$. We can then treat shared variables protected by locks as thread-local within critical sections. Let us attempt to use privatization to express the scheduling-sensitive analysis of Miné [2012] in constraint-based form.

$$\Lambda(x) \subseteq \mathsf{locks}(\rho_u) \qquad (\_, s, u) \in E_t, x \in \mathsf{write}_s \tag{8}$$

$$\rho_u(x) \sqsupseteq R_t(x) \qquad u \in N_t,\ \mathsf{locks}(\rho_u) \cap \Lambda(x) = \varnothing \tag{9}$$

$$G_t(x) \sqsupseteq \rho_u(x) \qquad (\_, s, u) \in E_t,\ x \in \mathsf{write}_s,\ \mathsf{locks}(\rho_u) \cap \Lambda(x) = \varnothing \tag{10}$$

$$\rho_v \sqsupseteq [\![s]\!]\rho_u \qquad (u, s, v) \in E_t \tag{11}$$

$$R_t \sqsupseteq G_{t'} \qquad t' > t \tag{12}$$

$$\rho_v \sqsupseteq G_{t'} \qquad (u, s, v) \in E_t,\ s \in \{\texttt{yield}, \texttt{lock}\},\ t' \neq t \tag{13}$$

This system is sound with respect to the concrete scheduling semantics of Miné [2012]; however, his abstract semantics is more precise as thread communication is propagated in and out of critical sections in a pair-wise fashion. If two threads protect communication via a common lock, but a third thread does not, we would no longer privatize this variable, thereby computing influences between all threads. A more faithful representation of the interference-based approach would be possible if we compute Rely-Guarantee invariants depending on the sets of held mutexes. This remains as future work.

On the other hand, we are more precise by only consider flow-insensitive influence from higher-priority threads, except at yield and locking instructions where a lower-priority thread can potentially influence a higher-priority thread. It is not clear to us why Miné [2012] ignores priorities in influences.

5 Weak Memory Consistency

These flow-insensitive abstractions are fairly imprecise with respect to the sequentially consistent semantics. Since many architectures provide weaker consistency guarantees, flow-insensitive abstractions are a good starting point for analyzing programs running on modern architectures. We previously claimed that our multithreaded analysis is sound with respect to the Linux kernel's memory model Vojdani et al. [2016], but we provided no rigorous proof. Suzanne and Miné [2018] have considered TSO and PSO models more rigorously. Our goal is to compare these abstractions as well as partial-order reduction techniques from model-checking for the analysis of concurrent programs communicating over weakly consistent memory.

References

P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In B. Beckert and R. Hähnle, editors, *Proc. of 2nd Int. Conf. on Tests and Proofs, TAP 2008*, volume 4966 of *Lecture Notes in Computer Science*, pages 116–133. Springer, 2008. doi: 10.1007/978-3-540-79124-9_9.

A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Log. Methods Comput. Sci.*, 8(1), article 26, 2012. doi: 10.2168/lmcs-8(1:26)2012

A. Miné. Relational thread-modular static value analysis by abstract interpretation. In K. McMillan and X. Rival, editors, *Proc. of 15th Int. Conf. on Verification, Model Checking, and*

Abstract Interpretation, VMCAI 2014, volume 8318 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2014. doi: 10.1007/978-3-642-54013-4_3

T. Suzanne and A. Miné. Relational thread-modular abstract interpretation under relaxed memory models. In S. Ryu, editor, *Proc. of 16th Asian Symp. on Programming Languages and Systems, APLAS 2018*, volume 11275 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2018. doi: 10.1007/978-3-030-02768-1_6

V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: The Goblint approach. In *Proc. of 31st ACM/IEEE Int. Conf. on Automated Software Engineering, ASE 2016*, pages 391–402. ACM, 2016. Press. doi: 10.1145/2970276.2970337

Towards Type-Level Model Checking for Distributed Protocols

Xin Zhao and Philipp Haller

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

Abstract

Developing correct distributed systems is notoriously difficult. Message-passing is a popular abstraction to simplify their implementation, supported by a number of projects including Akka and Orleans. However, the verification of these message-passing programs is still a challenge. There are recent works on type systems for verifying the behavior of message-passing applications but focused on the process/actor level, keeping system components such as transaction protocols unverified.

In this paper, we present preliminary work on ProtoMC, a type system that supports model checking for the correctness of distributed protocols. Our goal for ProtoMC is to compose processes verified using type-level model checking into protocols such that a well-typed program can be directly used as a verified implementation for a specific protocol.

1 Motivation

Model-checking is widely used in many distributed protocols such as 2PC, Paxos, and Raft. However, there might always exist a mismatch between the actual implementation and the protocol specification that is used for verification. There are works on how to automatically transform a specification to Implementation; however, due to the different choice of languages, a comprehensive approach remains elusive. Effpi [3] points out a type-based method that can specify and verify message-passing applications and ensure the type soundness of the program.

According to our experience in developing distributed systems, a more exciting target is to verify the implementation of specific protocols and even the composition of several protocol components. We want to extend Effpi to achieve the goal. However, the verification of complete protocols and their composition is challenging for three primary reasons: (1) Effpi focuses on verifying the safety/liveness properties of message-passing programs. [5] points out that 12 out of 15 projects they studied did not entirely stick to the Actor model; thus, the reasoning about message-passing programs should integrate with other programming and concurrency paradigms. (2) Effpi uses continuations to manipulate the "future" of the message passing, it increases the difficulty for developing a Hoare-stype logic reasoning. (3) Composition of verified protocols is first studied in Disel [4] in 2018 and the technique has improvement space.

2 The ProtoMC Language

ProtoMC is a programming language based on type-level model checking for distributed systems. In addition to Effpi, we add Hoare logic to check invariants guaranteed by protocols.

Example: a client-server system We write the following Effpi-style code to set up a client-server system where the server responds to client requests.

```
case class Req(key: Int, replyTo: ActorRef[Resp])
case class Resp(key: Int, res: Int, replyTo: ActorRef[Req])

def client(req: ActorRef[Req], key: Int): Actor[Resp] = {
  send(req, Req(key, self)) >>
  read { resp: Resp =>
    println("Result is: " + resp.res)
  }
}

def server(): Actor[Req] = {
  forever {
    read { req: Req =>
      send(req.replyTo, Resp(key,f(key),self))
    }
  }
}
```

In the above code, a client sends a `Req` message containing a request `key` and waits for the response from the server, and the server returns a response with the same `key`. Effpi can check the safety and liveness properties of the client-server system, e.g., whether the server responds to the client request or not; however, it cannot check whether the client receives the result for which it previously made a request, i.e., with a matching `key`. For each command, we create a Hoare triple $\{P\}S\{Q\}$ where P is the precondition, and Q is the postcondition. The variable pool is to record a set of requests that are going to be replied to, and m represents the received message. For example in client:

```
{pool = rs ∧ key ∈ dom(f)}
send(req, Req(key, self)) >>
{pool = (key, req) ⊎ rs}

{pool = (key, req) ⊎ rs ∧ m = Resp(key,res,req)}
  read { resp: Resp =>
    println("Result is: " + resp.res)
  }
{pool = rs }
```

We check the precondition for read statements such that the program is only correct if the received message contains the same key that appears in the pool. In this way, we are able to check the weak causality of the client protocol.

In order to integrate Hoare triples into a type system, we introduce *Hoare types* which augment types with pre- and postconditions.

Core ProtoMC. We propose the syntax for ProtoMC which extends Effpi with separation logic.

$$\begin{array}{rcll}
t & ::= & t\ t \mid \mathsf{let}\ x = t\ \mathsf{in}\ t' \mid \mathsf{chan}() \mid p \mid \ \ldots & \text{terms} \\
v & ::= & \lambda x.t \mid \mathbb{C} \mid \ \ldots & \text{values} \\
\mathbb{C} & ::= & \mathbb{a}, \mathbb{b}, \mathbb{c} \ldots & \text{channel objects} \\
p & ::= & \mathsf{end} \mid \mathsf{send}(t, t', t'') \mid \mathsf{recv}(t, t') \mid t || t' & \text{processes} \\
T & ::= & \{P\}\{Q\}\tau & \text{Hoare types} \\
\tau & ::= & \text{basic types} \mid \text{channel types} \mid \text{process types} & \text{basic } \mathsf{Effpi} \text{ types}
\end{array}$$

A Hoare type $\{P\}\{Q\}\tau$ is used to type a computation with a precondition P and a postcondition Q, computing a result of type τ.

We integrate Hoare types with Effpi's type system by extending the typing judgement with an additional predicate. Sent and Received are two auxiliary functions for calculating the state

$\{P\}$ and $\{Q\}$ after sending or receiving a message.

$$\frac{\Gamma \vdash \mathsf{send}(t_1, t_2, t_3) : \tau \quad \mathsf{Sent}(t_1, t_2, t_3, pr) \sqsubseteq (P, Q)}{\Gamma; pr \vdash \mathsf{send}(t_1, t_2, t_3) : \{P\}\{Q\}\tau} \text{ (T-SEND)}$$

$$\frac{\Gamma \vdash \mathsf{recv}(t_1, t_2) : \tau \quad \mathsf{Received}(t_1, t_2, pr) \sqsubseteq (P, Q)}{\Gamma; pr \vdash \mathsf{recv}(t_1, t_2) : \{P\}\{Q\}\tau} \text{ (T-RECEIVE)}$$

3 Challenges

We are still working on completing the semantics of ProtoMC. Here, we list some challenges for the design and implementation:

- A well-defined type system and complete soundness proof.
- Implementation of a verification system. We are currently devising the implementation of extended Effpi.
- Case studies for popular distributed protocols such as 2PC, Paxos, and Raft. The focus would be on creating state invariants for complex protocols.
- Experiments on the performance of such verification tools. We are considering the same approach as used in previous work, in order to evaluate the overhead and efficiency using well-known protocol implementations.

4 Related Work

Although session types have been studied for many years, Effpi [3] is one of the few systems that implement session types and build on top of a solid foundation. However, as we mentioned in the motivation section, the verification for only message passing is not enough for practical programs.

Diesel [4] is a type system that first studies the composition of the verified protocols. It gives us the motivation to extend Effpi with separation logic.

Actris [2] provides a functional correctness proof for concurrent programs with a mix of message passing, it is an extension of Iris project [1] which is a higher-order concurrent separation logic framework implemented and verified in the proof assistant Coq. [6] is another work based on Iris provides the first completely formalized tool for verification of concurrent programs with continuations.

References

[1] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program*, 28, article e20, 2018.

[2] Jesper Bengtson Jonas Kastberg Hinrichsen and Robbert Krebbers. Actris: Session-type based reasoning in separation logic. *Proc. ACM Program. Lang.*, 4(POPL), to appear.

[3] Alceste Scalas, Nobuko Yoshida, and Elias Benussi. Verifying message-passing programs with dependent behavioural types. In *Proc. of 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI 2019*, pages 502–516. ACM, 2019.

[4] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Proc. ACM Program. Lang.*, 2(POPL):28:1–28:30, 2018.

[5] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do scala developers mix the actor model with other concurrency models? In Giuseppe Castagna, editor, *Proc. of 27th European Conf. on Object-Oriented Programming, ECOOP 2013*, volume 7920 of *Lecture Notes in Computer Science*, pages 302–326. Springer, 2013.

[6] Amin Timany and Lars Birkedal. Mechanized relational verification of concurrent programs with continuations. *Proc. ACM Program. Lang.*, 3(ICFP):105:1–105:28, 2019.