AALBORG UNIVERSITET

# Proceedings of the 28th Nordic Workshop on Programming Theory (NWPT'16)


Kim G. Larsen and Jiří Srba

Aalborg University, Denmark


*Rold Storkro, North Jutland*

October 31 to November 2, 2016

# Preface

This volume contains the abstracs of talks that were selected for the presentation at the 28th Nordic Workshop on Programming Theory (NWPT'16). The workshop was held at Rold StorKro (North Jutland) in Denmark in the period October 31st to November 2nd, 2016. The event was organized by Kim G. Larsen and Jiri Srba with the support of Rikke W. Uhrenholt.

There were 32 submissions. Each submission was reviewed by three program committee members and the committee decided to accept 28 abstracts for the presentation. The program also included 4 invited talks by Andrzej Wasowski, Dines Bjørner, Jan Friso Groote and Alan Mycroft with the following abstracts (that appear in the order in which they were presented).

- **Finding Bugs in Linux Code with Type-And-Effect Abstraction** by *Andrzej Wasowski*, IT University of Copenhagen. Software projects suffer from conceptually simple resource manipulation bugs, such as accessing a de-allocated memory region, or acquiring a non-reentrant lock twice. The VBDB bug database contains entries for 100 such real bugs from several open source projects, including the Linux Kernel project. These historical bugs have been collected with the aim of giving concrete well understood and documented cases to program analysis researchers, in order to boost program verification research. I will discuss simplicity and complexity of real software manipulation bugs on examples selected from VBDB. One way to reduce the amount of such bugs is to use code scanners such as Smatch or Coccinelle. Unfortunately, while very efficient, code scanners are typically based on syntactic pattern matching, which is insufficient for identifying problems that span multiple functions and involve dynamically allocated memory. We have developed a shape-and-effect inference system for C that constructs a lightweight semantic abstraction, more analyzable than syntax. A model checker is then used to match semantic bug patterns over the control flow graph decorated with the shape-and-effect abstractions. Experiments run with our prototype analyzer (EBA) shows better precision and effectiveness than with syntactic bug scanners. We have been so far able to identify 10 previously unknown locking bugs in the Linux kernel. The bugs are confirmed as real by the Kernel developers, and five of them have been already fixed in response to our reports. I will conclude, sketching how we combine EBA with another tool, RECONFIGURATOR, to massively scan Linux kernel code for bugs in atypical source configurations. Joint work with Iago Abal, Claus Brabrand, Alexandru F. Iosif-Lazar, Aleksandar Dimovski, Jean Melo, and Stefan Stanciulescu.

- **Manifest Domains: Analysis & Description** by *Dines Bjørner,* Technical University of Denmark. Before software can be designed we must have a firm understanding of the requirements. Before requirements can be prescribed we must have a firm understanding of the domain. In this invited talk I discuss one approach to domain analysis & description in

which a "calculus" of analysis and description prompts are systematically applied by the domain analyzer cum describer "to the domain" (!) to yield a domain description. We outline this 'calculus'.

- **Using the modal mu-calculus for non functional properties** by *Jan Friso Groote,* Eindhoven University of Technology. The modal mu-calculus essentially consists of Hennessy-Milner logic. Within the context of mCRL2 we have extended it with data and time. The resulting logic is very expressive: for instance LTL can be translated linearly and verified without loss of efficiency. It is also very usable, allowing to formulate a wide range of functional properties about actual software systems in numerous application areas. An intriguing question is whether the modal mu-calculus can be used to model so called non functional properties, such as limiting behaviour or expected values. We show that when interpreted as real numbers, modal logic is nicely suited for this purpose.

- **Effects and Coeffects** by *Alan Mycroft*, University of Cambridge. Classic type systems essentially give types to procedures by giving the types of their parameters and their results. However, procedures can also have (side-) effects. These can be captured as part of the type of a procedure – giving an "effect system". More recently we showed how a dual notion of 'coeffects' can similarly express contextual or execution-environmental requirements for a procedure. Examples include security levels, linearity constraints, and access to hardware resources. These dual notions have associated dual semantic models as monads and comonads; these models give rise to program structuring mechanisms.

We thank the members of the PC for their work in selecting the talks. We also thank the invited speakers and all the participants (almost 50 of them) for making NWPT'16 a memorable event.

October 19th, 2016 — Kim G. Larsen and Jiri Srba
Aalborg University, Denmark

## Program Committee

| | |
|---|---|
| Lars Birkedal | Dept. of Computer Science, Aarhus University |
| Johannes Borgström | Uppsala University |
| John Gallagher | Roskilde University |
| Kim Guldstrand Larsen | Computer Science, Aalborg University |
| Dilian Gurov | KTH Royal Institute of Technology |
| Michael R. Hansen | Technical University of Denmark |
| Keijo Heljanko | Aalto University |
| Fritz Henglein | University of Copenhagen |
| Thomas Hildebrandt | IT University of Copenhagen |
| Anna Ingolfsdottir | Reykjavik University |
| Einar Broch Johnsen | University of Oslo |
| Yngve Lamo | Bergen University College |
| Alberto Lluch Lafuente | Technical University of Denmark |
| Fabrizio Montesi | University of Southern Denmark |
| Mohammadreza Mousavi | Halmstad University |
| Olaf Owe | Universitity of Oslo |
| Gerardo Schneider | Chalmers — University of Gothenburg |
| Cristina Seceleanu | Mälardalen University, Västerås, Sweden |
| Jiri Srba | Department of Computer Science, Aalborg University |
| Tarmo Uustalu | Institute of Cybernetics, Tallinn University of Technology |
| Juri Vain | Dept. of Computer Science, Tallinn University of Technology |
| Antti Valmari | Tampere University of Technology |
| Marina Waldén | Abo Akademi University |
| Uwe Wolter | University of Bergen |
| Wang Yi | Uppsala University |

## Table of Contents


Thread Summaries for Lock-Free Data Structures . . . . . 1
*Sebastian Wolff*

A Calculus of Virtually Timed Ambients . . . . . 4
*Einar Broch Johnsen, Martin Steffen and Johanna Beate Stumpf*

An operational semantics for Javascript DOM . . . . . 6
*Rasmus Thomas Tjalk-Boeggild and Christian W. Probst*

An Operational Framework for Multilevel Cache Coherent Multicore Architectures . . . . . 9
*Shiji Bijo, Einar Broch Johnsen, Ka I Pun and Silvia Lizeth Tapia Tarifa*

Multilevel Behavioural Metamodelling . . . . . 12
*Fernando Macias, Adrian Rutle and Volker Stolz*

What is your actual annotation overhead? . . . . . 15
*Duncan Cameron, Gudmund Grov and Leon McGregor*

A Sound Reasoning System Using Uninterpreted Predicates . . . . . 18
*Crystal Chang Din, Einar Broch Johnsen, Olaf Owe and Ingrid Chieh Yu*

On (de-)composing causality . . . . . 21
*Georgiana Caltais, Stefan Leue and Mohammadreza Mousavi*

From trash to treasure: timing-sensitive garbage collection . . . . . 24
*Mathias Vorreiter Pedersen and Aslan Askarov*

Demand-Driven Interprocedural Analysis for Map-Based Abstract Domains . . . . . 27
*Kalmer Apinis, Varmo Vene and Vesal Vojdani*

Towards Certified Compilation of Financial Contracts . . . . . 30
*Danil Annenkov and Martin Elsman*

A Secrecy-Preserving Language for Programming of Object-Oriented and Distributed Systems . . . . . 34
*Toktam Ramezanifarkhani and Olaf Owe*

Don't let data Go astray . . . . . 37
*Ka I Pun, Martin Steffen, Volker Stolz, Anna-Katharina Wickert, Eric Bodden and Michael Eichberg*

Integration of Runtime Verification into Metamodeling . . . . . 40
*Fernando Macias, Torben Scheffel, Malte Schmitz, Rui Wang, Martin Leucker, Adrian Rutle and Volker Stolz*

On Formalizing Information-Flow Control Libraries . . . . . 43
*Marco Vassena and Alejandro Russo*

Types for CAS: Relaxed Linearity with Ownership Transfer . . . . . . . . . . . . . . . 46
*Elias Castegren and Tobias Wrigstad*

Describing Behaviour Models through Reusable, Multilevel, Coupled Model Transformations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
*Adrian Rutle, Fernando Macias, Francisco Duran, Roberto Rodriguez-Echeverria and Uwe Wolter*

Pomset Languages of Higher-Dimensional Automata . . . . . . . . . . . . . . . . . . . . . . 52
*Uli Fahrenberg*

Approximating Probabilities in Static Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
*Maja Kirkeby*

An Operational Semantics for Multicasting Systems with Monotonic Values . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
*Seyed Hossein Haeri and Peter Van Roy*

Reductions for transition systems at work: the bisimulation hierarchy of state-to-function transition systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
*Marino Miculan and Marco Peressotti*

Finite tree automata determinisation and its application in program analysis and verification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64
*John P. Gallagher, Mai Ajspur and Bishoksan Kafle*

Delta-Oriented FSM-Based Testing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
*Mahsa Varshosaz and Mohammadreza Mousavi*

Enforcement and Specification of Evolving Privacy Policies for Social Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
*Raúl Pardo*

Non-interleaving opperational semantics for late and early Pi-calculus . . . . . 74
*Håkon Normann*

Minimizing Markov Chains Beyond Bisimilarity . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
*Giovanni Bacci, Giorgio Bacci, Kim Guldstrand Larsen and Radu Mardare*

wNetKAT: Programming and Verifying Weighted Software-Defined Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
*Kim Larsen, Stefan Schmid and Bingtian Xue*

Testing Software Product Lines using Differential Symbolic Execution . . . . . 83
*Sebastian Kunze and Mohammadreza Mousavi*

# Thread Summaries for Lock-Free Data Structures*

Sebastian Wolff

Fraunhofer ITWM, Kaiserslautern, Germany
Technische Universität Kaiserslautern, Germany
s.wolff@cs.uni-kl.de

It is an indubitable fact that lock-free data structures are unrivaled in terms of scalability on today's multi-core platforms. Unfortunately, they are also unrivaled in terms of complexity which is why automated verification is indispensable. However, static analyses suffer from poor scalability as they have to consider an unbounded number of client threads. This requirement is crucial since data structures are usually part of a library. Hence, one wants to verify the library for all possible usage scenarios once instead of verifying every usage.

In the following, we address this challenge. Additionally, we permit low-level memory interactions as known from languages like `C`. Currently, the most efficient verification technique to tackle this problem is so-called *thread-modular reasoning* [2, 1]. It abstracts the state space into a set of views. Each view captures information about an individual thread only – the relations among different threads is lost. In order to explore the state space two rules are applied exhaustively to generate new views from the existing ones. (1) Sequential rule: a view is changed by an action from the thread that it represents. (2) Interference rule: a victim view is updated according to the effect of a thread from another, interfering view taking an action. To do so the victim and interfering views are combined, the interfering thread takes an action and is finally projected away. This step is, however, problematic for two reasons: non-existent relations among threads might give false-positives and an exhaustive application requires to consider quadratically many pairs of views.

It is indeed this quadratic blow up during the state space exploration which makes thread-modular reasoning scale rather poorly [1] even though optimizations exist [4]. Intuitively, we want to avoid interference steps altogether in order to provide a truly scalable verification approach. Technically, we introduce *thread summaries* for this task and rephrase the verification problem in such a way that results for the modified problem carry over to the original one. Since this is work in progress we discuss our approach rather informally to convey the idea.

**Example** As an example consider Michael&Scott's queue the code of which is given in Listing 1. It maintains a singly-linked list with the pointers `Head` and `Tail` pointing to the first and last entry, respectively. The main problem introduced by lock-freedom is that adding a new entry to the list and swinging the `Tail` pointer to the newly inserted entry requires two separate actions which are not guarded from interfering threads. As a consequence, the data structure might appear malformed because of `Tail` not pointing to the last node. To resolve this issue every thread checks for this condition first and tries to *fix* the queue if needed.

In order to reason about the correctness of Micheal&Scott's queue one has to understand and exploit the underlying protocol. A protocol which is not unique to this implementation [3]. Figure 2 presents a modified version of the code which comprises the same protocol. However, the complexity of the actual implementation is avoided by *summarizing* the intended effect. That is, the original functions are rephrased as if they were executed atomically. It is worth noting that those *summaries* have properties quite beneficial for program verification. They avoid the read-copy-update cycle and the subtle interplay of threads. Moreover, they do not

*I would like to thank Lukáš Holík, Roland Meyer, and Tomáš Vojnar for the fruitful collaboration giving rise to the theory presented here. I would also thank Roland Meyer for the valuable comments on this paper.

```
/* shared */ pointer_t Head, Tail;
void enqueue(data_t v) {
  node = new Node(v);
  node.next = NULL;
  while (true) {
    tail = Tail;
    next = tail.next;
    if (tail != Tail) continue;
    if (next == NULL) {
      if (CAS(Tail.next, next, node))
        break;
    } else { CAS(Tail, tail, next); }
  }
  CAS(Tail, tail, node);
}
```

```
bool dequeue(data_t* v) {
  while (true) {
    head = Head;
    tail = Tail;
    next = head.next;
    if (head != Head) continue;
    if (head == tail) {
      if (next == NULL) return false;
      CAS(Tail, tail, next);
    } else {
      *v = head.data;
      if (CAS(Head, head, next)) {
        delete head;
        return true;
}}}}
```

Figure 1: Micheal&Scott's non-blocking queue, adapted from [5].

```
void $enqueue() {
  atomic {
    if (Tail.next != NULL) {
      Tail = Tail.next;
    } else {
      v = random_value();
      node = new Node(v);
      node.next = NULL;
      Tail.next = node;
}}}
```

```
void $dequeue() {
  atomic {
    if (Head == Tail) {
      if (Tail.next != NULL)
        Tail = Tail.next;
    } else {
      tmp = Head;
      Head = Head.next;
      delete tmp;
}}}
```

Figure 2: Thread summaries for Micheal&Scott's queue.

require any local state but determine their behavior by inspecting the shared heap. Our ambition is to use such summaries in order to enable more efficient static analyses.

**Abstract System** To overcome the poor scalability of static analyses for systems with an unbounded number of client threads we introduce a novel abstraction. Our *abstract* system is easier to analyse as it comes with just two concurrently operating threads: a *low-level* thread which executes the program code under scrutiny and a *high-level* thread which executes *thread summaries* like the ones presented above. Note that unlike before this system features two different types of threads.

The main contribution of our work is a theorem which guarantees that safety properties of the abstract system carry over to the concrete one. To that end, we instrument programs such that *bad states* can be identified inspecting the shared heap, e.g. by using a shared auxiliary variable. Thus, in order to verify safety properties it suffices that the reachable shared heap configurations of the proposed abstract system over-approximate those of the concrete system.

**Thread Summaries** In order to derive the desired theorem we have to establish a tight correspondence between the executions of the concrete and the abstract system. Intuitively, given some execution of the concrete system we pick an arbitrary thread as the low-level thread of the abstract system and replace every action of the other threads with summaries. We require those summaries to only mimic on the shared heap the actions that they replace. That is, they shall not modify the non-shared, local state of other threads, i.e. the low-level thread we just picked. It is worth noting that choosing summaries like that simplifies their task as they can ignore the local state of other threads. This results in fewer possibilities for summaries to react and thus gives a smaller search space during an analysis – a major step towards a scalable

approach. However, for a sound analysis we have to prove that no thread may ever influence the local state of another thread. We do so by enforcing a strict partitioning of the addresses in use into shared and owned ones. Moreover, thread summaries must not leak memory. Hence, cells owned by threads in the concrete system are freed in the abstract system if the owning thread is not present. Together with checking for malicious memory accesses, which we showed to be effective in previous work [4], we get adherence of both the high-level and the low-level thread to that partitioning. In particular, this means that no low-level thread will ever rely on cells that are owned by other threads in the concrete system as they appear freed in the abstract one.

Altogether, we come up with the following theorem which states the desired correspondence between abstract and concrete executions and allows a sound analysis of safety properties.

**Theorem.** *Let $S$ be some concrete system with low-level threads $t_1, \dots, t_n$ and let $S'$ be the corresponding abstract system with some low-level thread $t_i$ and high-level thread $t_s$. If the summaries executed by $t_s$ can (i) mimic every $t_i$ action on the shared heap, and (ii) are stateless, and if (iii) $S'$ adheres to the address partitioning, then*

$$reach_{shared}(S) \subseteq reach_{shared}(S') \quad and \quad reach_{local}(t_i, S) \simeq reach_{local}(t_i, S')$$

*where $reach_{shared}$ yields the reachable configurations of the shared heap, $reach_{local}$ gives the reachable configurations of the local heap and $\simeq$ denotes isomorphism up to dangling pointers.* □

Lastly, let us briefly discuss why we include a low-level thread in the abstract system at all. Why should we not just use a single high-level thread? In fact, keeping a low-level thread live in the abstract system allows one to verify whether or not the used thread-summary is correct, i.e. adheres to the requirements. This can be crucial. On the one hand, this enables refinement loops to be used. On the other hand, it allows for user-defined summaries while guaranteeing a sound analysis. Since we did not investigate how to effectively compute thread summaries yet the latter argument makes our approach practical even at this early stage.

**Experiments** We implemented a prototype[1] to check linearizability of lock-free data structures. The prototype is based on our previous tool which uses thread-modular reasoning [4]. We replaced interference steps with thread summaries and implemented all necessary checks to ensure their correctness. Our experiments prove our new verification technique to be truly superior to classical thread-modular reasoning in terms of scalability: while the analysis of Treiber's stack is only faster by factor 25, a more evident performance boost of factor 196 can be observed for Michael&Scott's queue. This substantiates the usefulness of our novel approach.

## References


[1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, volume 7795 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2013.

[2] J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, and S. Sagiv. Thread quantification for concurrent shape analysis. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008.

[3] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, volume 2180 of *Lecture Notes in Computer Science*, pages 300–314. Springer, 2001.

[4] F. Haziza, L. Holík, R. Meyer, and S. Wolff. Pointer race freedom. In *VMCAI*, volume 9583 of *Lecture Notes in Computer Science*, pages 393–412. Springer, 2016.

[5] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC*, pages 267–275. ACM, 1996.


[1] Available at: `https://github.com/Wolff09/TMRexp/tree/NWPT16`

# A Calculus of Virtually Timed Ambients

Einar Broch Johnsen, Martin Steffen, and Johanna B. Stumpf

University of Oslo, Oslo, Norway
{einarj, msteffen, johanbst}@ifi.uio.no

**Motivation.** The ambient calculus is the process algebra of locations and domains, originally developed by Cardelli and Gordon [2] for distributed systems such as the Internet. In this paper, we extend the ambient calculus with a notion of virtual time as a resource. The resulting calculus can be used for instance to model aspects of virtualization in cloud computing, where different locations, barriers between locations, and barrier crossing are important features, as well as elasticity which allows to provision virtual resources on-demand.

**Previous work on timed process algebras.** Algebraic concurrency theories such as ACP, CCS and CSP have been extended to deal with time-dependent behaviour in various ways (e.g., [1, 5, 3]). All these approaches describe speed as the absolute *duration* of processes, while in our approach speed describes the relative *processing power* of an ambient.

**Preliminaries on mobile ambients.** An ambient represents the location or domain where a process is running. Ambients can be nested, such that a surrounding *parental ambient* contains *subambients*, and the nesting structure can change dynamically. This is specified by three basic capabilities. The input capability $\mathit{in}\ n$ indicates the willingness of a process, respectivly its containing ambient, to enter an ambient named $n$, running in parallel outside, e.g., $k[\mathit{in}\ n.P] \mid n[Q] \rightarrow n[k[P] \mid Q]$. The output capability $\mathit{out}\ n$ enables an ambient to leave its surrounding ambient $n$, e.g., $n[k[\mathit{out}\ n.P] \mid Q] \rightarrow k[P] \mid n[Q]$. The third basic capability $\mathit{open}\ n$ allows to open an ambient named $n$ which is on the same level as the capability, e.g., $k[\mathit{open}\ n.P \mid n[Q]] \rightarrow k[P \mid Q]$. This syntax, as well as the semantics we consider, is based on [4] and largely unchanged compared to [2].

**Virtually timed mobile ambients.** We extend mobile ambients with notions of virtual time and resource consumption. Virtual time is a medium, which is made available to a location by its parental location, similar to time slices that an operating system provisions to its processes. Interpreting the locations of ambients as a place of deployment, each timed ambient is modelled to have a certain computing power, determined by its deployment. Thus, our model of timed ambients uses a *local* notion of time, which, however, is *relative* to the computing power of the embedding, parental ambients.

*Timed systems.* A timed ambient contains one *local clock* and possibly other timed ambients or classic untimed ambients and processes. A *computing environment* is a timed ambient which contains *resources*, as explained below.

*Local clocks.* To represent the outlined time model, each timed ambient is equipped with one local clock responsible for triggering timed behaviour and local resource consumption. Clocks have a *speed*, interpreted *relative* to the speed of the surrounding timed ambient. The speed $s$ of a clock is given by the rational number $p/q$, where $p$ is the number of local time slices emitted for a number $q$ of time slices received from the surrounding ambient. Time slices propagate from parental clocks to clocks in the subambients. Thus, the time in a nested ambient is relative to the global time, depending on the speeds of the clocks of the ambients it is nested in. We assume one universal outermost ambient with a *global clock* triggering the clocks of the local subambients recursively. When moving timed ambients, we must update the clocks to guarantee a correct propagation of time slices. As virtual time is made available to an ambient by its surroundings we have to ensure that a clock distributes time slices to all of its current subambients. Thus, we use an *update function* and define timed capabilities $\mathbf{in}\ n, \mathbf{out}\ n$, and $\mathbf{open}\ n$ for timed systems, corresponding to the similar untimed capabilites.

*Computing resources.* An ambient's processing power is defined by a *resource process* which transforms the time slices of the local clock into locally consumable resources. Processes expend the processing power of the ambient they are contained in by consuming resources. An ambient with a higher local clock speed produces more resources per parental time slice which in turn allows more work to be done for each parental time slice.

**Weak bisimulation for timed ambients.** We define weak timed bisimulation for virtually timed ambients as a conservative extension of weak bisimulation for mobile ambients as defined by Merro and Zappa Nardelli [4]. We then define a bisimulation for specific classes of processes which relaxes the condition on timing. This way we can determine if a system is faster than another and give a worst case approximation for this timing difference. We finally show that weak timed bisimulation for ambients completely characterises reduction barbed congruence for virtually timed ambients, extending a result from [4].

## References


1. J. C. M. Baeten and J. A. Bergstra. *Real Time Process Algebra* . Technical Report CS-R 9053, Centrum voor Wiskunde en Informatica (CWI), 1990.
2. Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Theor. Comput. Sci.,*, 240(1): 177–213, 2000.
3. Matthew Hennessy and Tim Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.
4. Massimo Merro and Francesco Zappa Nardelli. Behavioral theory for mobile ambients. *J. ACM*, 52(6):961–1023, November 2005.
5. Faron Moller and Chris Tofts. A temporal calculus of communicating systems. In J. C. M. Baeten and J. W. Klop, editors, *Proc. CONCUR'90*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer, 1990.

# An operational semantics for Javascript DOM

Rasmus T. Tjalk-Bøggild[1] and Christian W. Probst[2]

[1] Technical University of Denmark, Lyngby, Denmark
s103433@student.dtu.dk
[2] Technical University of Denmark, Lyngby, Denmark
cwpr@dtu.dk

## 1 Introduction

While Javascript itself and the semantics of the language have been studied in some detail [6], the Javascript DOM remains largely unstudied and is a large source of bugs in real life applications [5].

In this paper we develop an operational semantics for Javascript including the Document Object Model (DOM), as a major first step towards building a static analysis tool for detecting certain classes of Javascript DOM bugs. The operational semantics fully describe DOM Core Level 1 API [7] and we show how to extend this to include all of the DOM APIs.

## 2 Modelling the DOM

The DOM model we used is heavily inspired by the DOM model by Gardner *et al.* [4]. This DOM model relies on Context Logic, introduced by Calcagno et al [1], to represent the various DOM-trees. At its core, the DOM will contain a single rooted tree as the document. However, a single tree is not enough, as new elements to the document will be created outside the main document and inserted later. This feature requires a forest of rooted trees rather than a single rooted tree. To reflect the DOM structure, we will assign names to each node and names to collections of child nodes. To avoid ambiguity, we will call a named collection of trees *a forest* and call a collection of one or more forests and trees *a grove*.

Formally, using the approach from Gardner et al [4] [3], we have trees $t \in T$, forests $f \in F$, groves $g \in G$, attributes $a \in A$, strings $s \in S$, an infinite set of unique identifiers, $ID$ and a finite set of text characters $CHAR$ , with $id, fid, aid \in ID$ and $c \in CHAR$. Each element will be denoted by its type, id and children in the form: $TYPE_{id}[CHILDREN]$ where $TYPE$ is the element type (trees, strings, etc) and $CHILDREN$ denotes any other elements contained as children.

In our semantics we denote the entire DOM-heap as $D$; this is in effect the *doc* element with both the elements currently in the document and the grove containing the elements out of the document hierarchy.

To describe parts of the DOM model we introduce contexts for the various DOM structures. For each DOM structure we introduce a context that describe a rooted DOM structure of that type. This allows us to describe every legal DOM element (trees, nodes, strings, etc) as a context to reason about, which again enables local Hoare reasoning about the updates.

## 3 An operational semantics for Javascript DOM

The small-step semantics developed by Maffeis et al [6] provides us with a set of semantic functions for the core Javascript 1.5 [2] that operates on triples consisting of the global heap,

current scope and current term, but notably nothing about the underlying document. We extend this semantics by adding the DOM to the global state by including a global DOM-forest that contains all the DOM-trees that are in the context of the Javascript program. The DOM API provide the access points to the DOM-forest while the actual references to DOM-elements are still be stored in the heap - that is, the Javascript program will only be able to manipulate the DOM-forest through this API and we can therefore leave most of the Javascript semantics unchanged and focus only on the API-elements. Only these special API functions will have side-effects on the DOM, whereas the rest of the Javascript functions leaves the DOM unchanged between each operation:

$$\frac{H, I, e \xrightarrow{e} H_1, I_1, e_1 \qquad e \notin DOM\ API}{H, D, I, e \xrightarrow{e} H_1, D, I_1, e_1,}$$

Here $H$ denotes the heap, $D$ the DOM-forest (separate from the heap), $I$ the address of the current object and $e$ is the JS expression under evaluation.

### 3.1 Heap, values, and the DOM

In the Javascript small-step semantics we have the heap as the "permanent" store, which maps locations to objects and values which are being manipulated by the current expressions [6]. To extend the semantics to include the DOM, we introduce a structure for the DOM-forest in the semantics. The semantics then defines the DOM API in terms of how this modifies both the DOM-forest as well as the objects directly available to the Javascript code via the heap.

In terms of the semantic functions we keep the heap as a map from addresses to objects and introduce a DOM-function that maps IDs to DOM-trees. These IDs can be stored in the heap as a Javascript object. All manipulation of the DOM-structure will go via the API — even for DOM-nodes returned via the API. We are guaranteed that no element is ever deleted from the DOM heap (it may be removed from the main document or modified, but never actually deleted). We are therefore sure that once an ID has been created in the DOM that it will always exist when querying the DOM - it may not be part of the main document, but the API does not specify any way to actually delete DOM-objects.

## 4 Results and future work

To illustrate how the operational semantics for the minimal Javascript DOM works, we highlight an example from the API and describe in detail how this function works and how the semantics is constructed.

### 4.1 Example: Document.createElement

For the *document.createElement* API call, `r = d.createElement(n)`, the Javascript client provides three elements: the document id $d$, the name of the new element $n$ as a string, and the location for the new element id; all these elements are stored in the JS heap: In the DOM-heap, we assert that the provided document id, $d$, is in fact the id of the root node, there is a location in the heap, $r$, of unknown current value (to hold the new element id), there exists an empty grove (to insert the new element into) and that the provided name consists only of legal characters:

$$H1 := H(n = N, d = \&D, r = -), D1 := D(\#DOC_D, \emptyset_G) \wedge elmName(N)$$

The condition that an empty grove exists is trivially true, but the rest of the formula asserts that there is the correct connection between the heap IDs and the DOM-forest.

$$H1 := H(n = N, d = \&D, r = -), D1 := D(\#DOC_D, \emptyset_G) \wedge elmName(N)$$
$$\text{r = d.createElement(n)}$$
$$H1(r = \&eId), D1(\langle Node_{eId}[nodeName_{aId}[upperCase(N)] \odot defaultAttr(N), \emptyset_F]\rangle_G)$$

The resulting JS-heap now contains a new id, $eId$, at the location of $r$, which is the only change made there, whereas the DOM-heap now contains a new element with the $nodeName$ property set and any default properties available for elements of this type (the $defaultAttr$ assertion).

### 4.2 Future work

The semantics for Javascript DOM allows us to build proofs by hand to express for instance, that a given piece of Javascript code will never cause DOM exceptions. The next big step is to build a static analysis tool that uses the semantics to analyse for potential errors in the DOM API usage. However, Javascript code found in the wild usually uses quite a bit more than just the Core DOM Level 1 API, and a secondary task is therefore to expand the semantics to include more of the DOM APIs. Another interesting approach we are considering is to formalise the semantics in an automated theorem prover, to ease the development of proofs.

## References

[1] Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context logic and tree update. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 271–282, 2005.

[2] ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf, December 1999. Accessed: 2016-08-17.

[3] Philippa Gardner and Azalea Raad. DOM: Specification, Client Reasoning and Implementation, 2016.

[4] Philippa Gardner, Gareth Smith, Mark J. Wheelhouse, and Uri Zarfaty. Local Hoare reasoning about DOM. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2008, June 9-11, 2008, Vancouver, BC, Canada*, pages 261–270, 2008.

[5] Frolin S. Ocariza Jr., Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. An Empirical Study of Client-Side JavaScript Bugs. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, Maryland, USA, October 10-11, 2013*, pages 55–64, 2013.

[6] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An Operational Semantics for JavaScript. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, pages 307–325, 2008.

[7] Document Object Model Level 1 Core, Appendix E: ECMA Script Language Binding. https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/ecma-script-language-binding.html, October 1998. Accessed: 2016-03-26.

# An Operational Framework for Multilevel Cache Coherent Multicore Architectures *†

Shiji Bijo, Einar Broch Johnsen, Ka I Pun, S. Lizeth Tapia Tarifa

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

## 1 Motivation

Multicore architectures are gaining popularity in today's hardware design. Applications that are deployed on these architectures are expected to scale and perform better by using the available parallel resources. Language and compiler designers can benefit from the operational understanding of the low-level interactions between cores, caches and main memory during task execution, where performance depends from where data is fetched (e.g., caches or main memory). This paper presents work on an abstract formal model which captures multicore architectures with multilevel private caches in terms of operational semantics of parallel executions and an implementation of the proposed semantics in rewriting logic.

## 2 The Abstract Model

Our model captures cores in a multicore architecture as independent processing units which use caches to speed up their access to data from main memory. When a core requires data for reading or writing, it first tries to access the data from cache memory. If the data is found in the cache, we get a *hit*, otherwise we get a *miss* and need to bring a copy of such data from main memory. Data is stored in cache memory as cache lines together with its block address from main memory. If the cache memory is full, a block should be evicted to make space for the new block. Since multiple copies of data can co-exist in different caches and main memory, our model includes the standard MSI protocol which keeps all copies of data consistent. In the **MSI** protocol, a cache line will be in one of the three states: **M**odified, **S**hared, or **I**nvalid. A *modified* state indicates that the core has the most recently updated copy and that all other copies are *invalid*, while a *shared* state indicates that multiple copies can co-exist and are consistent.

Figure 1 illustrates the components of multicore architectures with single level caches. As described in papers [1] and [2], cores execute tasks consisting of (low-level) instructions, scheduled by a task queue. Cores exchange messages by broadcasting through a communication medium which abstracts from concrete topologies such as mesh, bus, or ring. Since communication between cores is order of magnitude faster than fetching data from main memory, we model the medium as instantaneous and use label matching on a process algebra style transition system to capture the communication. In case of a cache miss for a block with address $n$, a read message, captured by the label $!Rd(n)$, will be broadcast through the medium. All other components receive the message, captured by the dual label $?Rd(n)$. Upon receiving this message, a cache with a modified copy of the block flushes it to main memory and update the state to shared, so the sender can fetch the most recent copy from the main memory. To perform a successful write operation, the cache line's state should be modified or shared. An attempt to write to block $m$ in a shared cache line will broadcast an invalidation message (with label

*Supported by EU project FP7-612985 *UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimisations* (www.upscale-project.eu). †This work extends papers [1,2].

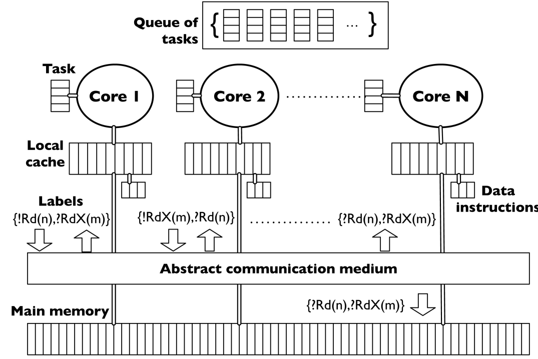



Figure 1: Abstract model of multi-core architecture (illustration) [1,2].

$!RdX(m)$). After receiving this message (using the dual label $?RdX(m)$), all components with block $m$ in shared state will mark it as invalid. A successful write operation marks the state of the new copy as modified.

The outlined abstract model will also work for architectures in which each core has multilevel private caches. In this case, the cache levels of a core are exclusive such that the intersection of block addresses for any two levels is always empty. This way data redundancy among the cache levels of a core is avoided, which makes it possible to store more blocks. For multilevel caches, a read request will be propagated through all caches of a core and it is only broadcast if the core cannot find the block in *shared* or *modified* state in any of its private caches.

# 3 Operational Semantics

We have extended the structural operational semantics (SOS) for cache coherent multicore architecture [1,2], with multilevel caches, which consists of both global and local transition rules. A configuration *Config* mainly consists of cores, caches and main memory. Let $\mathit{Config} \xrightarrow{*} \mathit{Config}'$ denote an execution starting from a configuration *Config* and reaching configuration $\mathit{Config}'$ by applying zero or more global transition rules, in turn triggering local transition rules.

**Global steps** capture interactions between caches and main memory to fetch and flush data, task scheduling, and the global protocol which guarantees data consistency. In a multicore environment, cores communicate concurrently and the synchronisation between cores is done in terms of labels $!Rd(n)$, $!RdX(n)$, $?Rd(n)$, and $?RdX(n)$. This synchronisation ensures data consistency and absence of data races. Rule Top-Synch-2 illustrates one of our global rules and shows how broadcast works with label synchronisation. The symbol $\circ$ separates components, over-lined terms $\overline{Ca}$ denote multi-sets of caches and $\overline{CR}$ multi-sets of cores, with a *belongs*-relation connecting caches to cores. The set $S$ consists of sending labels and $R$ of receiving labels. The rule captures the recursive decomposition of $S$ until dual labels are generated for each group of components, which consists of a core and its private caches.

$$\textsc{(Top-Synch-2)}\quad \frac{\begin{array}{c} \mathit{belongs}(\overline{Ca}_1, \overline{CR}_1) \quad \mathit{belongs}(\overline{Ca}_2, \overline{CR}_2) \quad S = S_1 \uplus S_2 \quad R_1 = \mathit{dual}(S_1) \quad R_2 = \mathit{dual}(S_2) \\ \overline{CR}_1 \circ \overline{Ca}_1 \xrightarrow{S_1 \cup R_2 \cup R} \overline{CR}'_1 \circ \overline{Ca}'_1 \quad \overline{CR}_2 \circ \overline{Ca}_2 \xrightarrow{S_2 \cup R_1 \cup R} \overline{CR}'_2 \circ \overline{Ca}'_2 \end{array}}{\overline{CR}_1 \circ \overline{Ca}_1 \circ \overline{CR}_2 \circ \overline{Ca}_2 \xrightarrow{S \cup R} \overline{CR}'_1 \circ \overline{Ca}'_1 \circ \overline{CR}'_2 \circ \overline{Ca}'_2}$$

**Local steps** capture the local transitions in main memory, the local executions of statements in each core, interactions among cores and first level caches or between adjacent caches and the local actions derived from the global protocol to keep local data coherent with respect to the other components. Rule $\textsc{PrWr}_1\text{-}\textsc{L}_1\text{-}\textsc{Hit}_2$ illustrates one of our local rules and captures the execution of a write statement in a core in which the required block $n$ is in shared state and the invalidation message is broadcast. The symbol $\bullet$ separates elements inside components. The rule requires that the label $!RdX(n)$ matches with one of the elements of the set $S$ at the global level. After the write operation, the updated copy is in modified state.

$$(\textsc{PrWr}_1\text{-}\textsc{L}_1\text{-}\textsc{Hit}_2)\quad \frac{\mathit{first}(Lev) = \mathit{true} \quad \mathit{Cid}(Lev) = c \quad n = \mathit{addr}(r) \quad \mathit{status}(M, n) = \mathit{sh}}{(c \bullet \mathtt{write}(r); \mathit{rst}) \circ (Lev \bullet M \bullet \mathit{dst}) \xrightarrow{!RdX(n)} (c \bullet \mathit{rst}) \circ (Lev \bullet M[n \mapsto \mathit{mo}] \bullet \mathit{dst})}$$

**Metatheory.** We show that this semantics correctly implements cache coherence for any program and any number of cores and cache levels, by guaranteeing properties such as data race free access to data and no access to stale data. Results will be discussed at the workshop.

**Implementation.** The proposed operational semantics has been implemented in Maude [3], a high-level declarative language for executable specifications based on rewriting logic. So far we use simulations to study how different data layouts in main memory can influence parallel computations with shared data, and reachability analysis to verify correctness properties related to data consistency. The model also includes counters for cache hit/miss ratio and penalty for data accesses per cache, to collect information about the performance and cost of data accesses during simulations.

## 4 Summary

Related work focuses on simulation tools for multicore programs (e.g., [6]), analysis of cache coherence protocols (e.g., [5]), or weak/relaxed memory models without caches (e.g., [4]), whereas our work proposes an abstract formalisation of program execution which considers data movement between caches. In previous work, we developed an SOS, metatheory, and implementation for multicore architectures with single level coherent caches [1, 2] which we here extend to multilevel caches. This framework allows the specification and comparison of program execution with different design choices for the underlying hardware architecture, such as the number of cores, cache levels, and cache line organisation, as well as data layout in main memory.

## References

[1] S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. An operational semantics of cache coherent multicore architectures. In *Proc. SAC'16*, pages 1219–1224. ACM, 2016.

[2] S. Bijo, E. B. Johnsen, K. I Pun, and S. L. Tapia Tarifa. A Maude framework for cache coherent multicore architectures. In *Proc. WRLA'16*. LNCS 9942, pages 47–63. Springer, 2016.

[3] M. Clavel *et al.* *All About Maude*, *LNCS* 4350. Springer, 2007.

[4] K. Crary and M. J. Sullivan. A calculus for relaxed memory. In *Proc. 42nd Annual ACM Symp. on Principles of Programming Languages (POPL 2015)*, pages 623–636. ACM, 2015.

[5] G. Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 23(3):257–301, 2003.

[6] J. E. Miller *et al.* Graphite: A distributed parallel simulator for multicores. In *16th Intl. Conf. on High-Performance Computer Architecture (HPCA-16)*, pages 1–12. IEEE Computer Society, 2010.

# Multilevel Behavioural Metamodelling

Fernando Macias, Adrian Rutle, and Volker Stolz

Bergen University College
first.last@hib.no

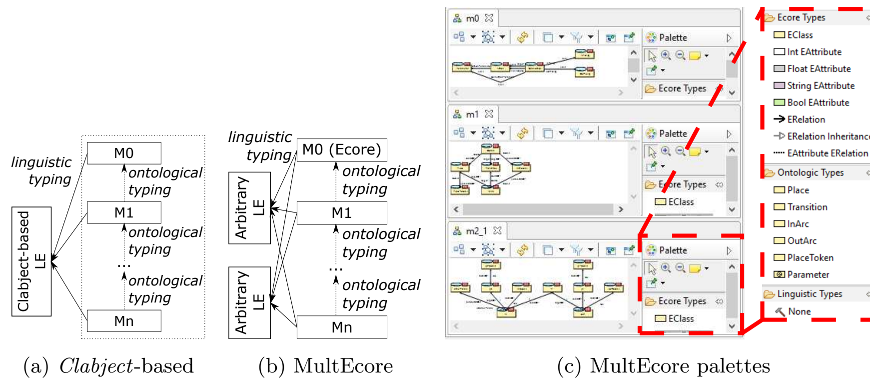
Languages such as Petri Nets, Flow Graphs and Finite State Machines are widely used to express software behaviour [12]. Behaviour metamodelling is a method to specify such languages by using (meta)models to define the syntax, and using model transformations to define the behaviour of these languages [6, 10], that is, the semantics. Metamodelling refers to the act of defining metamodels which specify the abstract syntax of a modelling language. The abstract syntax defines the set of modelling concepts, their attributes and their relationships, as well as the rules for combining these concepts to specify valid models. For instance, the Ecore metamodel from the Eclipse Modelling Framework (EMF) can be used to define a metamodel for Petri Nets, which in turn can be used create a specific Petri Net model. The creation of models with different levels of abstraction improves the clarity and reusability of behavioural models. However, mainstream metamodelling approaches like UML and EMF only provide a fixed number of user-accessible levels. This limitation forces users to mix levels of abstraction and create *synthetic* conformance relations among elements in the same model [1].

Multilevel modelling is gaining momentum as an improvement of the traditional fixed-level modelling [4]. This technique allows users to create arbitrarily big modelling stacks, with as many levels of abstraction as required. Hence, it favours modularization, model clarity and flexibility, which we believe are huge advantages in behavioural metamodelling. Using fixed-level metamodelling, like EMF, the available levels get occupied for, respectively, language definition and model specification. However, more levels are generally required in order to realise more advanced techniques, like executable modelling [3] and model transformation reuse [12]. For executable modelling, the bottommost instance is the deployed version of the modelled system, and it will evolve at runtime. The way this instance evolves, that is, its semantics or behaviour, is implemented through model transformations. These transformations have many commonalities among similar languages, such as Petri Nets and Coloured Petri Nets. Hence, being able to define these transformations based on an abstract model that comprises both types of Petri Nets leads to reuse for different kinds of models [12, 11].

The definition of conceptual frameworks for multilevel metamodelling is still an open debate (see MULTI 2016 CfP: `http://swt4.informatik.uni-mannheim.de/multi-2016/cfp.html`), which has led to different metamodelling frameworks. In this paper, we analyse them and propose some modifications which lead, as we argue, to increased flexibility and clarity of multilevel hierarchies. Furthermore, we present a tool that implements such a framework in EMF, avoiding the technology lock-in and steep learning curve that other approaches sustain.

## 1 Conceptual Frameworks for Multilevel Metamodelling

The first and most common approach to multilevel metamodelling is realised by creating a linguistic metamodel. This metamodel defines concepts such as `Level`, `Attribute` and `Clabject` [1]. These concepts can then be used to create levels that contain elements, and encode a *synthetic* typing relation (ontological) among elements in different levels. This way, the whole multilevel stack of models becomes an instance of the linguistic metamodel. Fig. 1a depicts this, where the whole stack (dotted block) is an instance of the model on the left. This




(a) *Clabject*-based (b) MultEcore (c) MultEcore palettes

Figure 1: Approaches to multi-level modelling (a and b) and MultEcore editors (c)

way of modelling, while common, poses two main problems. The first one is that the whole stack becomes dependent on one metamodel. Additionally, this causes incompatibility with EMF and UML tools, and forces the users towards custom-made tools, which do not offer the same capabilities, ecosystem and level of maturity. An enhancement which could avoid these challenges would be by considering Ecore as the linguistic metamodel, since this provides more expressiveness when creating the ontological stack. However, the user still needs to add explicit concepts for `Level` and a typing relation to make it usable.

As a consequence, we propose to place Ecore at the top of our ontological stack and exploit the concept of *potency* [9]. By using the right potencies, the user will be able to create instances of Ecore elements at any level, making the ontological stack independent from any linguistic metamodel. This approach has two direct benefits: (i) it aligns the conceptual framework with the internal implementation of the EMF, reducing technology lock-in, and (ii) the independence on linguistic metamodels allows for adding more expressiveness to the models in the ontological stack, an idea that can be used for a-posteriori typing [5], symbiotic languages [2] and runtime verification [8]. These extensions can be added incrementally and independently from one another. Fig. 1b displays the proposed enhancements used as background to implement MultEcore. And finally, we plan to take our proposal even further by exploiting the possibility of having a hierarchy of metamodels also in the linguistic side of the hierarchy. The core ideas summarised here are presented in more detail in [7].

## 2 Example in MultEcore

MultEcore is an EMF plugin that aides the creation of multilevel metamodelling stacks. It is based on the conceptual framework presented in the previous section. The goal of this plugin is to be as unintrusive as possible into the normal user experience of EMF, as a means of reducing the aforementioned technology lock-in. In Fig. 1c, we display several models in the same ontological stack, created and visualized using the default MultEcore editor. This editor, together with an Eclipse plugin that bypasses the restriction on the number of levels in EMF, is available for download at the MultEcore website `http://prosjekt.hib.no/ict/multecore/`.

One of the main advantages of MultEcore is its strong focus on EMF compatibility. Instead of creating a tool depending on a clabject-like linguistic metamodel and a set of custom-made

tools, our plugin reuses the EMF formats for models and metamodels. This way, the user can use any EMF-based tool required for any metamodelling-related task, such as code generation or model transformation. Furthermore, we provide an alternate Sirius-based[1] editor that is more oriented towards the creation of multilevel stacks, so that the user has yet another choice for her editor. MultEcore enhances EMF by allowing deep modelling stacks and multiple linguistic metamodels. From the user perspective, this is realized as a richer palette with the following types, listed in order of appearance in the palette displayed in Fig. 1c.

- **Ecore types** Having the Ecore metamodel with potency `1..*` at level M0 makes its types available at any level.
- **Ontological types** As in any modelling tool, the user can instantiate the metamodels. Furthermore, the use of potencies makes available some of the types from metamodels at higher levels of the hierarchy.
- **Linguistic types** Depending on the linguistic extensions connected to the ontological stack, some linguistic types can appear in the palette. These types can be added into already existing elements with an ontological type, as well as used to create brand-new instances with just the linguistic type. This functionality is still under development.

## References

[1] C. Atkinson and R. Gerbig. Flexible deep modeling with Melanee. In *Modellierung (Workshops)*, volume 255 of *LNI*, pages 117–122. GI, 2016.

[2] C. Atkinson, R. Gerbig, and B. Kennel. Symbiotic general-purpose and domain-specific languages. In *Intl. Conf. on Software Engineering*, pages 1269–1272. IEEE Press, 2012.

[3] C. Atkinson, R. Gerbig, and N. Metzger. On the execution of deep models. In *1st Intl. Workshop on Executable Modeling*, volume 1560 of *CEUR Workshop Proceedings*, 2015.

[4] J. de Lara, E. Guerra, and J. Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46, Dec. 2014.

[5] J. de Lara, E. Guerra, and J. Sánchez Cuadrado. A-posteriori typing for model-driven engineering. In *MoDELS*, pages 156–165. IEEE, 2015.

[6] A. H. Ghamarian, M. J. d. Mol, A. Rensink, E. Zambon, and M. V. Zimakova. Modelling and analysis using GROOVE. *IJSTTT*, 14(1):15–40, Feb. 2012.

[7] F. Macías, A. Rutle, and V. Stolz. MultEcore: Combining the best of fixed-level and multilevel metamodelling. In *3rd Intl. Workshop on Multi-Level Modelling*, CEUR Workshop Proceedings. CEUR-WS.org, 2016. To appear.

[8] F. Macias, T. Scheffel, M. Schmitz, and R. Wang. Integration of runtime verification into metamodeling for simulation and code generation. In *Intl. Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*. Springer, 2016.

[9] A. Rossini, J. de Lara, E. Guerra, A. Rutle, and U. Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, 26(6):1115–1152, 2014.

[10] A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, BM-FA '12, pages 5:1–5:10, New York, NY, USA, 2012. ACM.

[11] A. Rutle, F. Macias, P. Duran, R. Rodriguez, and U. Wolter. Describing Semantics of Behaviour Models through Reusable, Multilevel, Coupled Model Transformations. In *Nordic Workshop on Programming Theory*, November 2016. Submitted.

[12] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. In J. Cabot and E. Visser, editors, *Theory and Practice of Model Transformations, ICMT 2011*, pages 62–77. Springer, 2011.

[1] `http://www.eclipse.org/sirius/`

# What is your actual annotation overhead?[*][†]

Duncan Cameron, Gudmund Grov and Léon McGregor

Heriot-Watt University, Edinburgh, UK; {dac31,G.Grov,lm356}@hw.ac.uk

**Motivation** Users of modern *auto-active* program verifiers, such as Dafny [3], use the program text to encode: (i) the program specification, (ii) the program implementation, and (iii) the proof guidance used to ensure that the program satisfies the specification. Whilst both the implementation and the specification are desirable, the proof guidance is a necessary evil used only to guide the underlying prover, which for Dafny is the Z3 SMT solver [2].

The ratio of the number of lines of specification and proof annotations (LoA) to the number of lines of code (LoC) is called the *annotation overhead*. For non-trivial programs this ratio can be high – e.g. '5 LoA to 1 LoC'. Such a high degree of annotations may obfuscate the program text and act as a hindrance for mainstream uptake of the technique beyond niche markets.

However, there are at least two reasons why a stated annotations overhead may be too large:

1. Proof constructs are normally added incrementally to the program text until a proof is found. Previous increments may or may not have helped to progress the proof. Instead of attempting the (possibly time consuming and tedious) job of manually "tidy up" these annotations, a developer may be satisfied that the job is done.

2. The underlying verifier is improved such that guidance that used to be required is no longer needed. Periodically "tidying" annotations of existing libraries may be too time consuming for a developer, in particular if this has to be done manually.

Here, we investigate the degree in which annotations are superfluous. We develop a tool that automates the process of removing "dead proof annotations" of Dafny programs. This is applied to programs found in the Dafny library, and integrated with the Dafny Visual Studio IDE. Before going into these details, the types of proof constructs analysed are first outlined.

**Our approach** Dafny is an imperative, object-oriented and functional programming language that has been designed for verification. Properties that have to be satisfied are commonly specified by *contracts*: given a precondition that a method can assume, it must guarantee that a given postcondition holds. Many annotation and programming constructs are supported to provide guidance to ensure that the contract holds[1]. We will focus on the following constructs:

**Assertions** The simplest form of proof guidance is the use of **assert** statements in the code. An assertion must be verified, but can then afterwards be used by the prover.

**Lemma (calls)** Dafny supports a *ghost state* which is only used for verification and will not be compiled. A lemma is a ghost method, i.e. a method that lives in the ghost state. Lemmas can be used for more complex assertions that will require guidance to be proven. A lemma contract specifies the desired property, while the method body is used to encode the proof guidance (using programming and annotation constructs).

[*] This work has been supported by UK EPSRC grants EP/N014758 and EP/M018407. Thanks to Yuhui Lin, Vytautas Tumas and Rustan Leino for fruitful discussions.

[†] This is a short version of a paper developed in parallell for Fundamental Approaches to Software Engineering.

[1] For ease of reading we also use annotation for programming constructs used to guide proofs.

| Construct | #Progs | #Ints | % Removed | #∧ | % ∧removed |
|---|---|---|---|---|---|
| Assert | 68 | 322 | 88% | 50 | 16% |
| Lemma | 34 | 280 | 19% | – | – |
| Invariant | 39 | 255 | 15% | 327 | 18% |
| Variant | 37 | 97 | 20% | – | – |
| Calculation | 15 | 378 | 26% | – | – |

Table 1: Overview of results from applying DARe to the Dafny library.

**Loop invariants** Loop invariants express properties that have to hold throughout the execution of a loop, and are normally required for any program that contains a loop.

**Variants** All Dafny methods and loops must terminate. For simple cases, Dafny can prove this automatically; for more complex cases, the user must provide an expression, called a *variant*, which for each step has to decrease towards a lower bound.

**Calculations** Dafny supports Dijkstra style calculational proofs [4], where the user can encode each step of a calculation, possibly augmented by hints (e.g. lemma calls).

**Tool support & results**[2] We have developed a tool called DARe (Dead Annotation Removal) that works by traversing the Dafny abstract syntax tree and removing as many annotations as possible. Each time an annotation is removed, Dafny is applied to check if the program still verifies, and the constructs will only be removed if Dafny does not complain[3]. For an annotation that could not be removed, DARe will try to break up any conjunctions and remove as many conjuncts as possible[4].

Table 1 contains the results from applying DARe to 141 programs available from the Dafny library[5]. The second column ('#Progs') shows how many of the programs contained the given type of construct; '#Inst' shows the total number of instances of the construct; '% Removed' shows the percentage of how many constructs could be removed. Then of the remaining constructs that could not be removed: '#∧' shows the remaining conjuncts (after being broken up); and '% ∧removed' shows how many of these conjuncts could be removed. Note that for calculations, the table shows the sub-parts/steps of the calculations and not the full calculations themselves (albeit they will be removed if all elements of them are removed). Removing annotation had very little impact on the runtime of the verifier.

Dafny has a very modern Visual Studio based IDE [5], where the verification process happens in the background and is highlighted in the program text. This is comparable to how compilation errors are shown in systems like Visual Studio. A desirable feature is to integrate DARe with this plug-in and highlight annotations that can be removed. As this is an interactive system, *responsiveness* and *speed* are crucial; this can be at odds with guaranteeing the simplest possible program as illustrated by a program with the annotations:

```
assert A; assert B; assert C;
```

Assume that either the first assertion (A) or the second and third (B and C) are required. A naïve approach will delete **assert** A; and then be left with **assert** B; **assert** C; a complete approach will try all possible combinations and find that keeping **assert** A will lead to the simplest/shortest program. However, this will be very slow – we have to call Dafny for each

[2] More details available at: https://sites.google.com/site/tacnyproject/dare.
[3] We discuss a special case where we may want to keep an annotation even if Dafny does not complain below.
[4] E.g. if **assert** A ∧ B cannot be removed then DARe will try to simplify it to either **assert** A or **assert** B.
[5] Available at https://github.com/Microsoft/dafny/tree/master/Test.

step and attempting all possible removal combinations will not be sufficiently responsive. As an experiment, we implemented both versions for invariants and assertions, and found only one case where the simplest program gave a different result from the faster version for all 150 programs. In this case they still had the same number of annotations removed. We therefore used the fastest version, which has linear runtime. While we cannot formally guarantee that the shortest program is found, this experiment has shown that, in practice, it is likely to be.

Our Visual Studio DARe extension initially applies DARe to all methods (that does not have any verification errors) after the IDE has remained idle for at least 10 seconds. As even the faster version will be relatively slow to run, DARe will terminate (with failure) once a user starts interacting with the system. When it successfully terminates, all unnecessary/dead annotations will be highlighted (underlined and greyed out) as shown on line 31 of Figure 1 (top). The user can then press the light-bulb to get the options of removing the given annotation or all annotations of the method or file, as shown in Figure 1 (bottom). The tool will keep track of any methods that are changed since last time DARe was applied and will reapply DARe when a method have changed. Again, this will only happen if the method does not have any verification errors and the system remains idle for 10 seconds.

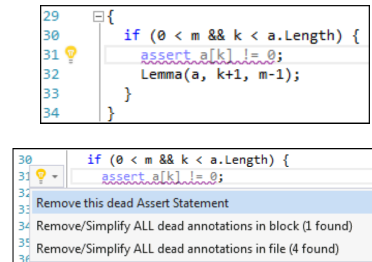



Figure 1: DARe IDE

**Conclusion & future work** The results have been encouraging and we are particularly surprised by the number of assertions that could be removed. The work is analogous to dead code elimination found in many compiler optimisations, but we are not familiar with any similar work for program verifiers, where the focus tend to be on the *discovery* of auxiliary annotations.

There are several ways to extend our work. We could address further simplification techniques, possibly normalising formulas to CNF form and then split the conjuncts. We could also extend the tool to other constructs. For example, programming constructs for the ghost state can also be removed. Another example is to remove/simplify contracts for (auxiliary) recursive methods which are analogous to loop invariants; in this case a user will need to indicate whether or not a lemma is auxiliary. Similarly, certain auxiliary lemmas that are no longer called can be removed. To improve runtime, we could integrate analysis of the proof terms generated by Z3 [2]: this could highlight constructs that are definitely needed or definitely not needed, which will help reducing the proof search. This may however be complicated by the way in which translation from Dafny into Z3 is handled via an intermediate verification language (called Boogie [1]). Finally, we could implement similar tools for other program verifiers and re-do the same experiment for them.

## References


[1] M. Barnett, B-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[2] Leo de Moura and Nikolaj Bjørner. Proofs and Refutations, and Z3. In *LPAR Workshops*, volume 418, pages 123–132, 2008.

[3] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.

[4] K. R. M. Leino and N. Polikarpova. Verified Calculations. In *VSTTE*, 2013.

[5] K. R. M. Leino and V. Wüstholz. The dafny integrated development environment. *arXiv preprint arXiv:1404.6602*, 2014.

# A Sound Reasoning System Using Uninterpreted Predicates

Crystal Chang Din, Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu

Department of Informatics, University of Oslo, Norway
{crystald,einarj,olaf,ingridcy}@ifi.uio.no

Software is involved in many different areas in the modern society and is constantly updated and extended. Code reuse and modular software development shorten time to market deployment and enhance the possibility to maintain complex software. Consequently, it is crucial to statically reason about each module before composing reused modules into products. However, a main complication of compositional reasoning with code reuse mechanisms, such as delta-oriented programming or class inheritance, is that the binding of method calls cannot always be decided at reasoning time. For instance in the delta-oriented setting, the binding depends on the order in which the delta modules are combined in a specific product. In particular, inside the definition of a method $m$ in a delta $D$ a call to *original* binds to the version of $m$ in the delta ordered before $D$ in the final product, which is not decided at the time $D$ is defined. In the setting of object-orientation and inheritance, the binding of methods is affected by method redefinitions in subclasses. Again the binding of calls cannot be decided before all subclasses are given. In the delta case, binding upwards is not known statically and in the case of inheritance binding downwards is not known.

In this paper we will explore a general rely-guarantee technique which is able to deal with compositional reasoning in both settings. This approach uses so-called *uninterpreted predicate variables*, i.e., providing symbolic names of the pre- and postcondition of methods for which the binding cannot be decided. Thus the pre- and postcondition of a defined method, with calls to such context-dependent methods, may contain occurrences of such uninterpreted predicate variables. For instance, in the setting of delta-oriented programming, consider a call to the *original* method with actual parameters $e$. One may then assume that the called method satisfies the Hoare triple $\{P\}\ original(e)\ \{Q\}$ where $P$ and $Q$ are simply symbols representing the actual pre- and post-conditions. When considering a specific product the binding of *original* calls can be decided, and one may then replace the uninterpreted symbols by the corresponding "concrete" predicates. This kind of reasoning is not restricted by the limitations inherent in the Liskov principle for Delta-oriented programming [4]. In the setting of inheritance, it gives a flexible way of defining method contracts, generalizing behavioral subtyping, lazy behavioral subtyping [3], and closed-world reasoning. Reasoning is modular while replacement of uninterpreted predicate symbols by concrete predicates may be gradually postponed, possibly until the class hierarchy is finalized.

The idea of *uninterpreted predicate variables* was used in [2]. However, it did not provide reasoning rules and soundness proof. The idea was also used to handle proof reuse [1]. Our contribution is a generalization, considering also inheritance and a soundness proof.

**Programming language and assertion language** We consider the object-oriented language Lightweight Java (LJ), https://rok.strnisa.com/lj/, which includes classes, fields, methods, assignment statements, if-else statements, object creation and method calls. The annotated method declarations `AMD` for our language are presented as the following:

$$
\begin{array}{lll}
\texttt{AMD} & ::= & \text{MD}\ \overline{sp} \\
sp & ::= & \textbf{guar}\ \{\overline{\texttt{ap}}\}\ \textbf{req}\ \{\overline{\texttt{req}}\} \\
\texttt{ap} & ::= & (a,a) \mid \textbf{readonly}\ \bar{\text{f}} \\
\texttt{req} & ::= & m : \{\overline{\texttt{ap}}\}
\end{array}
\qquad
\begin{array}{lll}
a & ::= & v \mid z \mid op(\overline{a}) \mid u \mid a\sigma \\
u & ::= & P \mid Q \mid ... \\
\sigma & ::= & \epsilon \mid [v_0, ..., v_i := a_0, ..., a_i](\text{where } 0 \leq i)
\end{array}
$$

| | |
|---|---|
| trivial | $m(\overline{x}) : (p, q),\ \Gamma \vdash m(\overline{x}) : (p, q)$ |
| internal | $\dfrac{\Gamma \vdash m(\overline{x}) : (p, q)}{\Gamma \vdash \{p[\overline{\mathrm{x}} := \overline{\mathrm{e}}]\}\ \mathrm{w} = m(\overline{\mathrm{e}})\ \{q[\overline{\mathrm{x}}, \mathtt{result} := \overline{\mathrm{e}}, \mathrm{w}]\}}$ |
| external | $\dfrac{\Gamma \vdash m(\overline{x}) : (p, q)}{\Gamma \vdash \{p[\mathtt{this}, \overline{\mathrm{x}} := v, \overline{\mathrm{e}}]\}\ \mathrm{w} = v.m(\overline{\mathrm{e}})\ \{q[\mathtt{this}, \overline{\mathrm{x}}, \mathtt{result} := v, \overline{\mathrm{e}}, \mathrm{w}]\}}$ |
| method | $\dfrac{\Gamma \vdash \{p[\overline{y} := \overline{y'}]\}\ \overline{\mathrm{s}}; \mathbf{return}\ e\ \{q[\overline{y} := \overline{y'}]\}}{\Gamma \vdash \{p\}\ N\ m(\overline{\mathrm{N}\ x})\{\overline{\mathrm{N}\ \mathrm{y}}, \overline{\mathrm{s}}; \mathbf{return}\ e\}\ \{q\}}$ |
| methodspec | $\dfrac{\Gamma \vdash \{p\}\ N\ m(\overline{\mathrm{N}\ x})\{\overline{\mathrm{N}\ \mathrm{y}}, \overline{\mathrm{s}}; \mathbf{return}\ e\}\ \{q\}}{\Gamma \vdash m(\overline{x}) : (p, q)}$ |
| methodU | $\dfrac{\Gamma \vdash \{p_i\}\ m(\overline{\mathrm{N}\ x})\{\overline{\mathrm{N}\ \mathrm{y}}, \overline{\mathrm{s}}; \mathbf{return}\ e\}\ \{q_i\}, \quad \text{for all } i\ .\ 0 \leq i \leq j, j \in \mathbb{N}}{m(\overline{\mathrm{N}\ x})\{\overline{\mathrm{N}\ \mathrm{y}}, \overline{\mathrm{s}}; \mathbf{return}\ e\}\ \mathbf{guar}\ \{\{(p_i, q_i) \mid 0 \leq i \leq j, j \in \mathbb{N}\}\}\ \mathbf{req}\ \{\Gamma\}}$ |

Figure 1: Selected reasoning rules for the UJ Language.

where MD is a method declaration, $\overline{sp}$ is a list of specifications, in which $\overline{\mathtt{ap}}$ are assertion pairs that the current method must guarantee, assuming that the requirements $\overline{\mathtt{req}}$ are satisfied by the methods invoked by the current method. An assertion $a$ may be a program variable $v$, a logical variable $z$, an expression $op(\overline{a})$ which applies an operator $op$ to a list of assertions, or a *uninterpreted predicate variable* $u$, which is conventionally capitalized, i.e., $P$ and $Q$. These uninterpreted assertions play the role of placeholders in symbolic assumptions and represent the pre- and postconditions of methods where the exact specifications are unknown at the reasoning time. Substitutions $\sigma$ bind program variables $v$ to assertions $a$. The assertion **readonly** $\overline{\mathrm{f}}$ declares variables $\overline{\mathrm{f}}$ as read-only. The use of this assertion restricts the set of assignable variables and is critical in the rely-guarantee setting.

Lightweight Java extended with `AMD` annotations is called UJ. We select a set of reasoning rules related to method calls for this language in Fig. 1. When analyzing a given method body the given guarantee specifications $(p_i, q_i)$ must be verified under the assumption of the stated requirements $\Gamma$ concerning other methods (including methods called in the body).

**Soundness** We prove soundness in the sense that provable results are valid results, i.e., implication ① in Fig. 2, where $\vdash_{\mathrm{UJ}}$ denotes the proof system for UJ and $\models_{\mathrm{UJ}}$ denotes validity of this system (The definition of validity is omitted here). Assumptions with uninterpreted predicate variables need to be interpreted by concrete specifications. This can be captured by replacing $\Gamma$ by a concrete instantiation without use of uninterpreted predicate variables, i.e., implication ② in Fig. 2, where $\tau$ is a substitution of uninterpreted predicate variables to concrete predicates. Thus $\Gamma\tau$ represents an "instantiation" of $\Gamma$ with all interpreted symbol replaced by the corresponding concrete predicates in $\tau$, and similarly for assertions $p\tau$ and $q\tau$. We let $\vdash_{\mathrm{LJ}}$ denote the Hoare-style reasoning systems for Lightweight Java with assertions without use of uninterpreted predicate variables and $\models_{\mathrm{LJ}}$ denote the standard notion of validity for such Hoare triples. We assume soundness of the reasoning system for LJ programs, which is based on well-known principles: $\Gamma \vdash_{\mathrm{LJ}} \{p\}\ \overline{s}\ \{q\} \Rightarrow \Gamma \models_{\mathrm{LJ}} \{p\}\ \overline{s}\ \{q\}$ for $\Gamma$, $p$ and $q$ without uninterpreted symbols. This assumption is applied in the proof as the implication ③ in Fig. 2. Soundness of $\vdash_{UJ}$ then reduces to the implication ④ in Fig. 2.

$$
\begin{array}{ccc}
\Gamma \vdash_{\mathrm{UJ}} \{p\}\, \bar{s}\, \{q\} & \Rightarrow_{①} & \Gamma \models_{\mathrm{UJ}} \{p\}\, \bar{s}\, \{q\} \\
\Downarrow_{④} & & \Uparrow_{②} \\
\Gamma\tau \vdash_{\mathrm{LJ}} \{p\tau\}\, \bar{s}\, \{q\tau\} & \Rightarrow_{③} & \Gamma\tau \models_{\mathrm{LJ}} \{p\tau\}\, \bar{s}\, \{q\tau\} \quad \text{for each } \tau
\end{array}
$$

Figure 2: Sketch of the soundness proof

**Applications** An implementation of class, `Account`, and its modified version, `UpdatedAccount`, in which the method `update` invokes the original `update` method in `Account`, is presented below:

```
class Account{
  int bal = 0; // the balance
  bool update(int x){
    bal = bal + x; return true;} }
```

```
class UpdatedAccount{ int limit = 0;
  bool update(int x){ bool b = false;
    if(bal+x > limit){b=original(x);}
    return b;} }
```

Note that `UpdatedAccount` can be tailored to a subclass for class inheritance or a modified class in deltas. We prove the following specifications of the `update` method in the `Account` class:

```
guar{(bal==bal', (bal==bal'+x)∧result)}
```

which specifies that the result of `update` increases the balance `bal` by `x`. The specifications of the `update` method in `UpdatedAccount` class is presented below:

```
guar{((bal+x>limit)∧P,Q)} req{original:{(P,Q)}}
guar{(bal==bal' ∧ (bal+x<=limit), bal==bal' ∧ ¬result)}
guar{readonly limit} req{original: {readonly limit}}
```

The first assertion specifies that if the balance increased by `x` is larger than the limit and if the precondition `P` of the invoked `original` holds, then after executing `update` the postcondition `Q` of the `original` method is true. The second assertion says if the balance increased by `x` is not larger than the limit, than the balance will not be changed by the `update` method. Besides, the field `limit` can neither be changed by the current method nor the original one. We prove that the `update` method in `UpdatedAccount` satisfies all three assertions, and the validity of proof result is preserved after substituting `P` and `Q` with concrete properties.

To conclude, we provide a sound reasoning system using uninterpreted variables which can be applied to modular code reuse such as class inheritance, delta- and traits-oriented programming. The presented approach should be well suited for semi-automatic verification in the KeY-system.

## References


[1] R. Bübel, F. Damiani, R. Hähnle, E. B. Johnsen, O. Owe, I. Schaefer, and I. C. Yu. Proof repositories for compositional verification of evolving software systems. *Transactions on Foundations for Mastering Change (FoMaC)*, to appear, 2016.

[2] F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A transformational proof system for Delta-oriented programming. In *16th International Software Product Line Conference, SPLC '12, Salvador, Brazil - September 2-7, 2012, Volume 2*, pages 53–60. ACM, 2012.

[3] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *The Journal of Logic and Algebraic Programming*, 79(7):578 – 607, 2010.

[4] R. Hähnle and I. Schaefer. A Liskov principle for Delta-oriented programming. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 5th International Symposium, 2012, Part I*, volume 7609 of *LNCS*, pages 32–46. Springer, 2012.

# On (De-)composing Causality*

Georgiana Caltais[1], Stefan Leue[1], and Mohammad Reza Mousavi[2]

[1] Department for Computer and Information Science, University of Konstanz, Germany
[2] Centre for Research on Embedded Systems, Halmstad University, Sweden

**Abstract**

This work introduces a notion of counterfactual causality in the Halpern and Pearl sense that is compositional with respect to the interleaving of non-communicating transition systems. The underlying logic is Hennessy Milner logic. This is an abstract based on a paper accepted in CREST 2016.

**Introduction.** Determining and computing causalities is a frequently addressed issue in the philosophy of science and engineering. A notion of causality that is frequently used in relation to technical systems relies on counterfactual reasoning [17]. In short, the counterfactual argument defines when an event is considered a cause for some effect, in the following way: a) whenever the event presumed to be a cause occurs, the effect occurs as well, and b) when the presumed cause does not occur, the effect will not occur either. Counterfactual arguments are the basis for a number of fault analysis, failure localization and software debugging techniques, such as delta debugging [24], nearest neighbor queries [21], counterexample explanation in model checking [10, 9] and why-because-analysis [13]. The seminal paper [11] describes an event model and a notion of actual causation encompassing the counterfactual argument. Most relevant for our work are the contributions in [16, 15]. The latter provide an interpretation of the results in [11] in the context of transition systems and trace models for concurrent system computations.

*The objective of this paper* is to consider the notion of counterfactual causality reasoning and actual causation in the context of labeled transition systems (LTS's). LTS's represent system models and Hennessy Milner logic (HML) [12] formulae specify the system properties for whose violation actual causes are sought. We also establish first results on computing causalities in this setting using (de-)compositional verification. This is an abstract of the paper [3], accepted in CREST 2016[1]. For the complete definitions and proofs we refer the interested reader to [3].

**Preliminaries.** Next, we provide a brief overview of LTS's and their computations, and HML. A *labeled transition system* (LTS) is a triple $T = (\mathbb{S}, s_0, A, \rightarrow)$, where $\mathbb{S}$ is the set of states, $s_0 \in \mathbb{S}$ is the initial state, $A$ is the action alphabet and $\rightarrow \subseteq \mathbb{S} \times A \times \mathbb{S}$ is the transition relation. We write $\twoheadrightarrow \subseteq \mathbb{S} \times A^* \times \mathbb{S}$, to denote the reflexive and transitive closure of $\rightarrow$. Two isomorphic LTS's $T$ and $T'$ are denoted by $T \simeq T'$.

Let $\mathcal{D}$, $\mathcal{D}_i$ range over possibly infinite lists of words in $A^*$; we write $\varepsilon$ for the empty word. We say that such lists are *size-compatible* if they are finite lists of the same length, or if they are all infinite lists. Let $\pi = (s_0, l_0, \mathcal{D}_0), \ldots (s_n, l_n, \mathcal{D}_n), s_{n+1} \in (\mathbb{S} \times A \times [A^*])^* \times \mathbb{S}$. Assume that $\mathcal{D}_0, \ldots, \mathcal{D}_n$ are size-compatible. Intuitively, we write $traces(\pi)$ to denote the pairwise extensions of $l_0 \ldots l_n$ with words "at the same level" in $\mathcal{D}_0, \ldots, \mathcal{D}_n$. For instance, if $\pi = (s_0, l_0, [w_1^0, w_2^0]), (s_1, l_1, [w_1^1, w_2^1]), s_2$, then $traces(\pi) = \{l_0 w_1^0 l_1 w_1^1,\ l_0 w_2^0 l_1 w_2^1\}$. We say that $\pi$ is a *computation* of $T$ whenever the following hold: (i) $s_0 \xrightarrow{l_0} s_1 \ldots \xrightarrow{l_n} s_{n+1}$, (ii) $\mathcal{D}_0, \ldots, \mathcal{D}_n$

*The work of Georgiana Caltais was partially supported by an Independent Research Start-up Grant founded by Zukunftskolleg at Konstanz University. The work of Mohammad Reza Mousavi has been partially supported by the Swedish Research Council (Vetenskapsrådet) award number: 621-2014-5057 (Effective Model-Based Testing of Concurrent Systems) and the Swedish Knowledge Foundation (Stiftelsen för Kunskaps- och Kompetensutveckling) in the context of the AUTO-CAAS HöG project (number: 20140312).

[1] https://crest2016.inria.fr

are size-compatible, and (iii) for all $w \in traces(\pi)$ there exists $s \in \mathbb{S}$ such that $s_0 \overset{w}{\twoheadrightarrow} s$. $sub(\pi)$ stands for the set of all computations $\pi' = (s_0, l'_0, \mathcal{D}'_0), \ldots, (s_m, l'_m, \mathcal{D}'_m), s'_{m+1}$ such that $l'_0 \ldots l'_m$ is a sub-word of $l_0 \ldots l_n$.

We consider formulae in *Hennessy-Milner logic* (HML) [12] given by the following grammar:

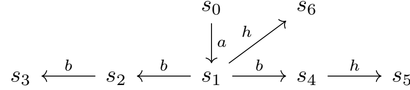
$$\phi, \psi ::= \top \mid \langle a \rangle \phi \mid [a]\phi \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \qquad (a \in A).$$

We say that an HML formula $\phi$ as above is *built over* $A$. The associated satisfaction relation $\vDash$ is defined in the standard way, over states $s \in \mathbb{S}$ and HML formulae. We call formulae $\phi$ that hold in the initial state of a system *immediate effects*.

**Defining causality.** Our notion of causality complies with that of "actual causation" proposed in [11] and further adapted to the setting of concurrent systems in [15]. Consider a transition system $T = (\mathbb{S}, s_0, A, \rightarrow)$; *causal traces* for an HML property $\phi$ in $T$ denoted by $Causes(\phi, T)$ is the set of all computations $\pi = (s_0, l_0, \mathcal{D}_0), \ldots, (s_n, l_n, \mathcal{D}_n), s_{n+1}$ such that

1. $s_0 \xrightarrow{l_0} \ldots s_n \xrightarrow{l_n} s_{n+1} \wedge s_{n+1} \vDash \phi$ *(Positive causality)*
2. $\exists \chi \in A^*, s' \in \mathbb{S} : s_0 \overset{\chi}{\twoheadrightarrow} s' \wedge s' \vDash \neg\phi$ *(Counter-factual)*
3. $\forall \chi' = l_0\chi_0 \ldots l_n\chi_n \in \{l_0 \ldots l_n\} \cup (A^* \setminus traces(\pi)),\ s' \in \mathbb{S} : s_0 \overset{\chi'}{\twoheadrightarrow} s' \Rightarrow s' \vDash \phi$ *(Occurrence)*
4. $\forall \chi' \in traces(\pi) \setminus \{l_0 \ldots l_n\},\ s' \in \mathbb{S} : s_0 \overset{\chi'}{\twoheadrightarrow} s' \Rightarrow s' \vDash \neg\phi$ *(Non-occurrence)*
5. $\forall \pi' \in sub(\pi) : \pi'$ *does not satisfy items 1. – 4. above (Minimality)*

Consider, for an example, the following LTS and the HML formula $\phi = \langle h \rangle \top$:




Item 1 above suggests that action $a$ should be a cause for $\phi$. Item 2 indicates that the hazard formula $\phi$ does not hold trivially everywhere as, for instance, $s_0 \xrightarrow{abb} s_3$ and $s_3 \nvDash \phi$. Item 3 states that $ab$ cannot be considered a cause as it can non-deterministically lead to $s_2$ and $s_3$ and only $s_4 \vDash \phi$ holds. Item 4 states that $(s_0, a, [\varepsilon]), s_1$ is not a cause for $\phi$ because extending $a$ with $bb$, for instance, violates $\phi$ and thereby violates item 3. However, $(s_0, a, [h, bb, bh]), s_1$ is a cause, because $a$ leads to a hazard, all possible extensions of $a$ with anything but $h$, $bb$ or $bh$, the only ones being $\varepsilon$ and $b$, also keep the hazard. Item 5 states that $(s_0, a, [\varepsilon, \varepsilon]), (s_1, b, [h, b]), s_4$ is not a cause because it is not minimal. This is because its sub-computation $(s_0, a, [h, bb, bh])$, $s_1$ is a cause as previously discussed.

**(De-)composing causality.** A *causal projection* of $T = (\mathbb{S}, s_0, A, \rightarrow)$ with respect to a property $\phi$, is $T' = (\mathbb{S}', s_0, A, \rightarrow')$ such that $\mathbb{S}' = \{s_i \mid 0 \leq i \leq n{+}1 \wedge (s_0, l_0, \mathcal{D}_0), \ldots, (s_n, l_n, \mathcal{D}_n), s_{n+1} \in Causes(\phi, T)\}$ and $\rightarrow' = \{(s_i, l_i, s_{i+1}) \mid 0 \leq i \leq n \wedge (s_0, l_0, \mathcal{D}_0), \ldots, (s_n, l_n, \mathcal{D}_n), s_{n+1} \in Causes(\phi, T)\}$. We write $T \downarrow \phi$ to denote the causal projection of $T$ with respect to $\phi$. Intuitively, a causal projection is an LTS whose executions capture precisely all causal traces.

Theorem 1 and Theorem 2 below show that reasoning on causality with respect to disjunctions and, respectively, conjunctions of HML formulae in the context of interleaved LTS's can be reduced to reasoning on causality in the corresponding interleaved components. We consider standard notions of interleaving ($||$) and non-deterministic ($+$) choice between LTS's [18].

**Theorem 1.** Consider LTS's $T = (\mathbb{S}, s_0, A, \rightarrow)$ and $T' = (\mathbb{S}', s'_0, B, \rightarrow')$ such that $A \cap B = \emptyset$. Assume two HML formulae $\phi$ and $\psi$ over $A$ and $B$, respectively. Whenever $\phi$ and $\psi$ are not immediate effects, the following hold:

$$\begin{array}{rcl} T \mid\mid T' \downarrow (\phi \vee \psi) & \simeq & T \downarrow \phi + T' \downarrow \psi \\ T \mid\mid T' \downarrow (\phi \wedge \psi) & \simeq & (T \downarrow \phi) \mid\mid (T' \downarrow \psi). \end{array}$$

**Discussion.** Causality for concurrency in the context event structures representing causal relationships between events was addressed, for instance, in [19]. (De-)compositional verification has been studied in contexts such as model-checking [2, 4, 23] and model-based conformance testing [20, 22]. Our approach is based on our earlier work on decompositional verification of modal mu-calculus formulae [1]. Regarding compositional verification of causality, we are only aware of the line of work in [7, 5, 6, 8]. Our initial results only concern interleaving components, but our long-term vision is that modal decomposition will enable mechanized decomposition of the modal formula for communicating components, following the approach of [14, 1].

**References**


[1] L. Aceto, A. Birgisson, A. Ingolfsdottir, and M. Mousavi. Decompositional reasoning about the history of parallel processes. In *FSEN 2011*, volume 7141 of *LNCS*, pages 32–47. Springer, 2012.

[2] H. R. Andersen. Partial model checking (extended abstract). In *LICS*, pages 398–407, 1995.

[3] G. Caltais, S. Leue, and M. Mousavi. (De-)Composing Causality in Labeled Transition Systems. Technical Report soft-16-02, 2016.

[4] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Component verification with automatically generated assumptions. *Autom. Softw. Eng.*, 12(3):297–320, 2005.

[5] G. Gößler and L. Astefanoaei. Blaming in component-based real-time systems. In *EMSOFT 2014*, pages 7:1–7:10. ACM Press, 2014.

[6] G. Gößler and D. Le Métayer. A general framework for blaming in component-based systems. *Sci. Comput. Program.*, 113:223–235, 2015.

[7] G. Gößler, D. Le Métayer, and J. Raclet. Causality analysis in contract violation. In *RV 2010*, volume 6418 of *LNCS*, pages 270–284. Springer, 2010.

[8] G. Gößler and J. Stefani. Fault ascription in concurrent systems. In *TGC*, volume 9533 of *LNCS*, pages 79–94. Springer, 2016.

[9] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3), 2006.

[10] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN*, LNCS 2648, pages 121–135. Springer, 2003.

[11] J. Halpern and J. Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British Journal for the Philosophy of Science*, 2005.

[12] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, 1980, Proceedings*, volume 85 of *LNCS*, pages 299–309. Springer, 1980.

[13] P. Ladkin and K. Loer. Analysing aviation accidents using wb-analysis – an application of multimodal reasoning. In *AAAI Spring Symposium*. AAAI, 1998.

[14] K. G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *J. Log. Comput.*, 1(6):761–795, 1991.

[15] F. Leitner-Fischer and S. Leue. Causality checking for complex system models. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *LNCS*, pages 248–267. Springer, 2013.

[16] F. Leitner-Fischer and S. Leue. Probabilistic fault tree synthesis using causality computation. *International Journal of Critical Computer-Based Systems*, 4:pp. 119–143, 2013.

[17] D. Lewis. *Counterfactuals*. Blackwell Publishers, 1973.

[18] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[19] H. Normann, C. Prisacariu, and T. T. Hildebrandt. Concurrency models with causality and events as Psi-calculi. In *ICE 2014*, volume 166 of *EPTCS*.

[20] N. Noroozi, M. Mousavi, and T. Willemse. Decomposability in input output conformance testing. In *MBT 2013*, volume 111 of *Electr. Proc. in TCS*, pages 51–66, 2013.

[21] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *ASE*, Canada, 2003.

[22] T. Villa, N. Yevtushenko, R. Brayton, A. Mishchenko, A. Petrenko, and A. Sangiovanni-Vincentelli. *The Unknown Component Problem, Theory and Applications*. Springer, 2012.

[23] G. Xie and Z. Dang. Testing systems of concurrent black-boxes—an automata-theoretic and decompositional approach. In *FATES*, volume 3997 of *LNCS*, pages 170–186. Springer, 2006.

[24] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Elsevier, 2009.

# From trash to treasure: timing-sensitive garbage collection

Mathias Pedersen and Aslan Askarov

Aarhus University, Aarhus, Denmark

## 1 Introduction

Secure execution of third party code is crucial in mobile applications, web browsers, cloud technologies, etc. It has long been known that static analysis in insufficient for the detection of side-channel attacks (e.g., cache attacks, power attacks, etc.). This paper shows that automatic memory management also exposes side-channels that can be used to leak sensitive information in third party code. We present a series of simple attacks on modern runtimes, in particular Java sequential and parallel garbage collections and V8 default garbage that illustrate the potential of the attack. These attacks have also been shown to be effective over a network connection.

To address the problem of leaks via garbage collection, we develop a model of a programming language that uses abstract secure runtime and security types to enforce security.

## 2 Attacking JVM and V8

We start with two general amplifiable timing attack strategies that exploit the garbage collector in order to leak one bit of information. Both attacks work for two garbage collection strategies used by Java, as well as for the generational mark-sweep/mark-compact strategy used in V8.

When a value is public we say that it is a low (L) value, and high (H) when the value is secret.

**High dependency in low context** The attack in the figure below exploits that, during evacuation from from-space to to-space, the amount of bytes copied depends on the reachable nodes at the current point in the program. Thus, by creating a sufficiently large difference in reachable and unreachable nodes, the time required to perform a minor/major garbage collection becomes observable.

The example leaks whether $h > 0$ by observing the time difference caused by the allocation on Line 9. If the value of *diff* is large then $h > 0$, and otherwise $h \leq 0$.

Suppose constants `size` and `size2` are chosen to satisfy the constraints i) $2 \cdot$ `size` $\leq$ $allocThresh$, and ii) `size` $+$ `size2` $\geq$ $allocThresh$, where $allocThresh$ is an experimentally obtained heap size that triggers garbage collection.

Assume the allocation on Line 9 invokes the garbage collector. If $h > 0$ there are two distinct arrays that need to be copied from from-space to to-space, meaning that $2 \cdot size$ integers are copied. However, if $h \leq 0$ then only the array allocated on Line 1 is reachable, and thus should be copied. Thus, the collector only copies *size* integers. This difference in the number of bytes that should be copied creates an observable difference in timing.

```
1  int[] a = new int[size];
2  int[] b = null;
3  if (h > 0) {
4    b = new int[size];
5  } else {
6    b = a;
7  }
8  long t1 = System.nanoTime();
9  int[] d = new int[size2];
10 long t2 = System.nanoTime();
11 long diff = t2 - t1;
```

**Low modification in high context** Consider the program to the right, and let constants `size` and `size2` be chosen so that i) `size2` $\leq$ $allocThresh$, and ii) `size` $+$ `size2` $\geq$ $allocThresh$. If $h > 0$ the allocation on Line 2 partially fills up the heap, so that the allocation on Line 6 requires a garbage collection, and thus diff is large. However, if $h \leq 0$ then no garbage collection occurs at Line 6, as the size of the memory does not exceed the implementation's threshold for GC, resulting in a small value for `diff`.

```
1  if(h > 0) {
2    int[] b = new int[size];
3    b = null;
4  }
5  long t1 = System.nanoTime();
6  int[] c = new int[size2];
7  long t2 = System.nanoTime();
8  long diff = t2 - t1;
```

## 2.1 Amplification of the attacks

We amplify the leakage of the one-bit–attacks described in Section 2 by repeating the attack for each bit in the secret. We present the results of applying this attack to leak the value of a 32-bit integer.

Figure 1 shows the output of running the program described in the previous section on the secret input $5342121$ with the serial garbage collection strategy used by Java.

Figure 2 shows the output obtained by running a similar attack on the V8 JavaScript engine using Node.js. This attack follows the same pattern as the attack on Java and has therefore been omitted.

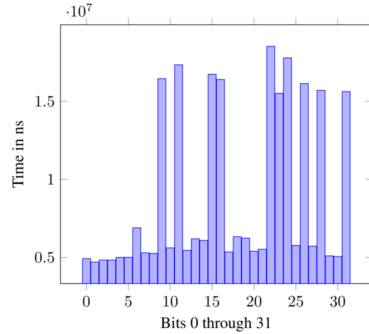



Figure 1: Leakage in serial collection strategy

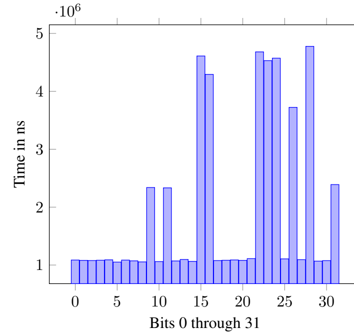



Figure 2: Leakage in collection strategy in V8

# 3 Formal semantics for secure garbage collection

In order to design a secure collection strategy, we develop a formal semantics that specifies how garbage collection is allowed to affect the heap; we study this in a setting of a small imperative language. Our main insight is that in order to prevent the types of attacks presented in the previous section, we must constrain which parts of the heap can be collected and when. When runtime program counter is low, only low parts of the heap can be collected; when the runtime program counter is high, only high parts of the heap can be collected. This is necessary because garbage collection represents a bi-directional information channel, which we explain using two examples inspired by our experiments from Section 2.

## 3.1 Motivating security restrictions on GC

**Implicit flows when collecting L in H** Consider the program in Figure 3, where we assume that $N$ and $M$ are constants. The non-standard command at $H$ with bound $v$ do $c$ ensures that the execution of the body of the at block takes exactly $v$ steps or diverges. Assume that $v$ is sufficiently large to bound the execution of the at command argument.

```
1  y := new_L (N, 0);
2  y := null;
3  at H with bound v do c {
4    if (h > 0) x := new_H(M,0)
5    else skip
6  }
7  t1  := time();
8  y   := new_L (N,0);
9  t2  := time();
10 low := t2 - t1
```

Figure 3: Implicit flow when collecting $L$ in $H$

```
1  x := new_H (M, 0);
2  at (H, v) {
3    if (h > 0) x := null
4    else skip
5  }
6  t1  := time();
7  y   := new_L (N,0);
8  t2  := time();
9  low := t2 - t1
```

Figure 4: Implicit flow when collecting $H$ in $L$

Because the high conditional is guarded by an at command, the value of $t_1$ does not depend on which branch of the conditional is taken. However, if the semantics of garbage collection allows low parts to be collected inside at, say before executing the allocation on Line 4, then $t_2 - t_1$ is likely to be short. This motivates that garbage collection should not collect low allocations when the program counter level is high.

**Implicit flows when collecting H in L** The example in Figure 4 shows that collecting high allocations when the program counter is low is also dangerous. Suppose we are given constants $M$, $N$, and $v$ as described earlier, and consider program below.

As before, the timing of the high conditional is protected with an at command. Consider allocation on Line 7 that may trigger garbage collection. If semantics of the GC allows collecting high allocation in the low program counter, the amount of time that the collector spends here will depend on whether $x$-array can be reclaimed, affecting the value of $t_2$. This motivates that garbage collection should not collect high allocations when the program counter is low.

### 3.2 Formal semantics for garbage collection

The rule below describes the formal garbage collection transition in the small-step semantics of our language. We write $h \rightsquigarrow^m_\delta h'$, when $h$ is a (sub)heap and $h'$ is the result of collecting in $h$ that takes time $\delta$, given memory $m$. Given a heap $h$ and a level $\ell$, write $h^{=\ell}$ for the heap that includes all allocations tagged with security level $\ell$. Similarly, $h^{\neq\ell} = h \setminus h^{=\ell}$.

$$\frac{\mathsf{reach}(m, h_1 \uplus h_2) \cap \mathsf{dom}(h_2) = \emptyset \quad {h_2}^{\neq pc} = \emptyset \quad h_1 = {h_1}^{\neq pc} \uplus {h_1}^{=pc} \quad {h_1}^{=pc} \uplus h_2 \rightsquigarrow^m_\delta {h_1}^{=pc}}{\langle c, pc, m, h_1 \uplus h_2, t\rangle \dashrightarrow \langle c, pc, m, h_1, t + \delta\rangle}$$

## 4 Security guarantees

For garbage collection in isolation we obtain a property of timing-sensitive noninterference. The noninterference property is possibilistic [3]: if memory $m$ and heap $h$ are low-equivalent to memory $s$ and heap $w$, then for a GC-transition with $(m, h)$, that takes some time $\delta$ there exists a GC-transition with $(s, w)$ that takes times $\delta$ and yields low-equivalent resulting heap and memory.

For program noninterference we use the notion of attacker knowledge [1, 2]. The attacker knowledge is the set of possible memories that are consistent with the memory after a sequence of program transitions. Using a standard information flow type system, we bound the attacker knowledge set from below by the set of all possible memories. This ensures that an attacker observing low values only does not learn anything about the secret input. Combining this noninterference result with the possibilistic noninterference result of our garbage collection, we obtain possibilistic program noninterference.

## 5 Conclusion

This paper presents a series of examples that demonstrate feasibility of information leaks via garbage collection. We conclude that in order to effectively control such leaks a tight integration between runtime and the source-level language are needed. We observe that despite drastic simplifications in the design of the language that simplifies difficult aspects such as direct timing attacks, closing leaks via garbage collector requires strong assumptions from the implementations.

## References

[1] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[2] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

[3] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symp. on Security and Privacy*, pages 177–186, May 1988.

# Demand-Driven Interprocedural Analysis for Map-Based Abstract Domains

Kalmer Apinis, Varmo Vene, and Vesal Vojdani

Institute of Computer Science, University of Tartu
J. Liivi 2, EE-50409 Tartu, Estonia
{kalmera,varmo,vesal}@ut.ee

## 1 Introduction

When applying static analysis to check specific properties in a given setting, such as data-race freedom in Linux device drivers, the task is greatly simplified by combining a series of textbook analyses [5]. Any serious analysis of C or other language with function pointers requires precise points-to information for those pointers. The complete results of such helper analyses, however, are not particularly interesting to users who primarily care about their specific properties, e.g., data-race warnings. Therefore, it would be ideal if inessential information would not be calculated. In fact, analysis frameworks already possess the means for demand-driven computation: *local solving*. This is used to solve the infinite constraint systems that arise from the data flow problem for inter-procedural analysis. Such systems can be solved with a local solver in a demand-driven fashion — starting from a relevant set of constraint system variables [2].

The goal of this work is to investigate possibilities of making constraint-system–based analyses even more demand-driven in the local solving framework. We propose a generic mechanism for cases where the analysis domain is a map structure or a function, e.g., points-to analysis that maps variables to their points-to sets, such that only values for those keys will be computed which are needed. Of the two classical approaches [4], the proposed idea has potential to improve in the call-string approach to inter-procedural static program analysis. In the functional approach, however, all map keys will be needed for analyzing function calls.

We have a proof-of-concept implementation for the idea based on the Goblint analysis framework [1], where we found that we can reduce the number of variables that are evaluated and save memory, allowing us to analyze larger programs; however, the helper analyses need to be optimized for the new framework in order to achieve reduction in the elapsed time for our race detection analysis.

## 2 Distributing map domains to separate variables.

A constraint system is a set of inequations that may contain variables from a set $\mathsf{Var}$. A *solution* to an inequation system is a variable assignment for which all inequations are satisfied. In static program analysis, the *left-hand side* of a single inequation is always just a single variable whereas the *right-hand side* is an expression possibly containing other variables, and the domain of the constraint system $\mathbb{D}$ has the structure of a *complete lattice*. Thus, constraint systems can be represented by functions $f \in (\mathsf{Var} \to \mathbb{D}) \to \mathsf{Var} \to \mathbb{D}$ in the sense that $f\,\sigma\,x$ computes the value of the right-hand side for the left-hand side $x$ and variable assignment $\sigma$.

We can distribute *any* mapping $\mathbb{D} = X \to \mathbb{D}_1$ from a set $X$ to a complete lattice $\mathbb{D}_1$ by currying $(V \to X \to \mathbb{D}_1) \to V \to X \to \mathbb{D}_1$ into its equivalent $((V \times X) \to \mathbb{D}_1) \to (V \times X) \to \mathbb{D}_1$. This produces a constraint system with variables $\mathsf{Var} = V \times X$ and the domain $\mathbb{D}_1$, where values for some $x \in X$ can be solved separately from other elements of $X$.

One source of map domains arises when information is stored for each program variable, e.g., constant propagation or points-to analysis — which we used to evaluate the framework. We first investigated the applicability of the idea to the functional approach to interprocedural static analysis. We found that function calls will generate dependencies to the complete state before the call — making the approach unsuitable for our goals. Applying the idea to the call-string approach, as we will see, is more successful. Consider the functions in a program as control-flow graphs, where $E$ is the set of labeled edges, program points $s_g$ and $r_g$ are the distinct starting and returning points, respectively. Using abstract transfer functions for statements and guards $[\![\cdot]\!]^\sharp$; along with abstract transfer functions for function calls $\mathsf{enter}^\sharp$ and $\mathsf{comb}^\sharp$ (as in [1]), the constraint system can be written down as follows:

$$\begin{array}{rll} \mathcal{C}[s_{\mathsf{main}}, \varepsilon, a] \sqsupseteq d^\sharp_{\mathsf{start}}\ a & & \\ \mathcal{C}[s_g, (u,v) :: c, a] \sqsupseteq \mathsf{enter}^\sharp\ \mathcal{C}[u,c]\ a & (u, g(), v) \in E & \\ \mathcal{C}[v, c, a] \sqsupseteq [\![l]\!]^\sharp\ \mathcal{C}[u,c]\ a & (u, l, v) \in E & (\mathcal{C}') \\ \mathcal{C}[v, c, a] \sqsupseteq \mathsf{comb}^\sharp_l\ \mathcal{C}[u,c]\ \mathcal{C}[r_g, (u,v) :: c]\ a & (u, g(), v) \in E\,. & \end{array}$$

Constraint system variables are written in the form $\mathcal{C}[u, c, a]$ where $u$ is a program point, $c$ the call-string, and $a$ the program variable; the syntax $\mathcal{C}[u, c]$ denotes partial function application and is to be read as $(\lambda b.\ \mathcal{C}[u, c, b])$. The first two constraints take care entering a function call based on whether the abstract call-string is empty $\varepsilon$ and we are entering the $\mathsf{main}$ function, or the call-string is not empty and we entered from some program point $u$. For demand-driven analysis, we must ensure that no unneeded dependencies will be generated. In this case, the value of a program variable at function entry only depends on its value at the call site.

The next constraint deals with non-call edges. Although the function $\mathcal{C}[u, c]$ is passed as the argument to $[\![l]\!]$, it is only used to query different values $\mathcal{C}[u, c, b]$. Thus, no unneeded dependencies need to be generated by $[\![l]\!]$. The last constraint handles the returning from a function call. No extra dependencies arise, as the function $\mathsf{comb}_l$ has only to decide if the value of variable $a$ comes from the called function $\mathcal{C}[r_g, (u, v) :: c, a]$, from before the call $\mathcal{C}[u, c, a]$, or some combination of the two.

## 3 Experimental evaluation

We implemented the distributed constraint system along with the combined constraint system on top of the Goblint static analyzer framework which analyses multi-threaded programs written in the C language.[1] The modifications needed for this paper were extensive as, first, Goblint used the functional approach, and second, transfer functions had to be curried manually. Only very light testing was performed and no optimization was done. Thus, the resulting data should be considered as preliminary.

The first program that we analyzed is the mathematical simulation code 433.milc (su3imp) taken from SPEC CPU2006 benchmark suite [3]. Solving the combined constraint system reached the partial solution in about 2.3 minutes — giving values to 141,280 constraint system variables. As the analysis domain is a mapping, Goblint computed 16,896,112 values. Solving the distributed constraint system took about 6.1 minutes and resulted in the partial solution containing values for 5,322,855 constraint system variables. This means that we only needed to compute about 32% of all values, showing that it is theoretically possible to significantly reduce computational effort and memory footprint. The number of constraint variables, however,

[1] Available from https://github.com/kalmera/analyzer

increased about 2.6 times and analysis time increased about 2.7 times. The correlation of these numbers is not surprising as the implementation of transfer functions does the same amount of work in the uncurried as in the curried version. It is unclear by how much the curried version could be optimized. In the current setting where reachability must be computed for every program point it is impossible for the distributed constraint system to compute less constraint variables than the combined constraint system. This need not necessarily be that way — the number of computed program variables would be significantly smaller if the analysis would assume that all program points were eventually reachable.

A similar evaluation was done in multi-threaded setting where we analyzed the Linux device driver lp: the generic parallel printer driver. Solving the combined constraint system for `lp.c` required computation of 2,138 constraint system variables which incorporated values for 24,253 program variables whereas solving the reachability using the distributed constraint system needed to solve 9,677. The combined constraint system was faster, taking only 0.6 seconds as solving the distributed constraint system took about 2.1 seconds. The overall result is similar to the single-threaded case: the new approach needed about 38% of values, computed 4.5 times more constraint system variables, and it took 3.5 times more time.

## 4 Conclusion

We have proposed and investigated a mechanism for making interprocedural static analysis more demand-driven by distributing map domains between different constraint system variables. Although the experimental evaluation was not extensive, we can see that the mechanism works in general. It is, however, a practical challenge to write the transfer function in such a way to keep the overhead of distributing map keys to different constraint system variables small. In our preliminary experiments, the overhead made the mechanism impractical as it produced an analysis time increase. However, as less information (60% fewer variables) is computed, also less computer memory would be required. To get better results one can sacrifice precision, e.g., consider non-deterministic branching instead of evaluating all conditional guards, to achieve fewer dependencies. In any case, more evaluation is needed to understand the performance characteristics of such trade-offs.

**Acknowledgement.** This work was funded by institutional research grant IUT2-1 from the Estonian Research Council.

## References

[1] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *APLAS*, pages 157–172. LNCS 7705, Springer, 2012.

[2] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In *IFIP Conf. on Formal Description of Programming Concepts*, pages 237–277. North-Holland, 1977.

[3] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[4] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Application*, pages 189–233. Prentice-Hall, 1981.

[5] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In *ASE*, pages 391–402. ACM, 2016.

# Towards Certified Compilation of Financial Contracts

Danil Annenkov and Martin Elsman

University of Copenhagen
Dept. of Computer Science (DIKU)
{daan,mael}@di.ku.dk

**Abstract**

We present an extension to a certified financial contract management system that allows for templated financial contracts and for integration with financial models through verified compilation into so-called payoff-expressions, which readily allow for determining the value of a contract in a given evaluation context, such as contexts created for simulations. The templating mechanism is useful both at the contract specification level, for writing generic reusable contracts, and for reuse of code that, without the templating mechanism, needs to be recompiled for different evaluation contexts. We report on the effect of using the certified system in the context of a GPGPU-based Monte Carlo simulation engine for pricing various over-the-counter (OTC) financial contracts. The full contract-management system, including the payoff-language compilation, is verified in the Coq proof assistant and certified Haskell code is extracted from Coq and integrated with an efficient OpenCL pricing engine.

## 1 Background and Motivation

New technologies are emerging that have potential for seriously disrupting the financial sector. In particular, blockchain technologies, such as Bitcoins [7] and the Ethereum Smart Contract peer-to-peer platform [9], have entered the realm of the global financial market and it becomes essential to ask to which degree users can trust that the underlying implementations are really behaving according to the specified properties. Unfortunately, the answers are not clear and errors may result in irreversible high-impact events.

The work presented here builds on a series of previous work on specifying financial contracts [1, 2, 4, 6, 8] and in particular on a certified financial contract management engine and its associated contract DSL [3]. This framework allows for expressing a wide variety of financial contracts (a fundamental notion in financial software) and for reasoning about their functional properties (e.g., horizon and causality). As in the previous work, the contract DSL that we consider is equipped with a denotational semantics, which is independent of stochastic aspects and depends only on an *external environment* $\mathsf{ExtEnv} : \mathbb{N} \times \mathsf{Label} \to \mathbb{R} + \mathbb{B}$, which maps observables (e.g., the price of a stock on a particular day) to values. In the work presented here, we present a certified compilation scheme that compiles a contract into a *payoff function*, which aggregates all cashflows in the contract, after discounting them according to some model. The result represents a single "snapshot" value of the contract. The payoff language, which is inspired by traditional payoff languages and is well suited for integration with Monte Carlo simulation techniques for pricing, is essentially a small subset of a C-like expression language enriched with notation for looking up observables in the external environment. We show that compilation from the contract DSL to the payoff language preserves the cashflow semantics.

The contract DSL described in [3], deals with concrete contracts, such as a one year European call option on the AAPL (Apple) stock with strike price $100. The lack of genericity means that each time a new contract is created (even a very similar one), the contract management engine needs to compile the contract into the payoff language and further into a target language for embedding into the pricing engine. To avoid this recompilation problem, we introduce the notion of a *financial instrument*, which allows for templating of contracts and which can be

turned into a concrete contract by instantiating template variables with particular values. For example, a European call option instrument has template parameters such as maturity (the end date of the contract), strike, and the underlying asset that the option is based on. Compiling such a template once allows the engine to reuse compiled code, giving various parameter values as input to the pricing engine.

Moreover, an inherent property of contracts is that they evolve over time. This property is precisely captured by a contract reduction semantics. Each day, a contract becomes a new "smaller" contract, thus, for pricing purposes, contracts need to be recompiled daily, resulting in a dramatic compile time overhead. To avoid recompilation in this case, the generated payoff code is parameterized over the *current time* so that evaluating the payoff code at time $t$ gives us the same result (upto discounting) as first advancing the contract to time $t$, then compiling it to the payoff code, and then evaluating the result.

The contract analysis and transformation code forms a core code base, which financial software crucially depends on. A certified programming approach using the Coq proof assistant allows us to prove the above desirable properties and to extract certified executable code.

## 2 The Contract Language

We assume a countably infinite set of program variables, ranged over by $v$. Moreover, we use $n$, $i$, $f$, and $b$ to range over natural numbers, integers, floating point numbers, and booleans. We use $p$ to range over parties. The contract language that we consider follows the style of [3] and is extended with template variables:

$$c ::= \texttt{zero} \mid \texttt{transfer}(p,p) \mid \texttt{scale}(e,c) \mid \texttt{translate}(t,c) \mid \texttt{checkWithin}(e,t,c,c) \mid \texttt{both}(c,c)$$

$$e ::= op(e,\ e,\ \ldots,\ e) \mid \texttt{obs}(l,i) \mid f \mid b \qquad t ::= n \mid v \qquad op ::= \texttt{add} \mid \texttt{sub} \mid \texttt{mult} \mid \texttt{lt} \mid \texttt{neg} \mid \texttt{cond} \mid \ldots$$

Expressions ($e$) may contain *observables*, which are interpreted in an external environment. A contract may be empty (`zero`), a transfer of one unit (for simplicity) (`transfer`), a scaled contract (`scale`), a translation of a contract into the future (`translate`), the composition of two contracts (`both`), or a generalized conditional $\texttt{checkWithin}(cond,t,c_1,c_2)$, which checks the condition *cond* repeatedly during the period given by $t$ and evaluates to $c_1$ if $cond = \texttt{true}$ or to $c_2$ if *cond* never evaluates to `true` during the period $t$.

The main difference between the original version of the contract language and the version presented here is the introduction of *template expressions* ($t$), which, for instance, allows us to write contract templates with the contract maturity as a parameter. This feature requires refined reasoning about the temporal properties of contracts, such as causality. Certain constructs in the original contract language, such as $\texttt{translate}(n,c)$ and $\texttt{checkWithin}(cond,n,c_1,c_2)$, are designed such that basic properties of the contract language, including the property of causality, are straightforward to reason about. In particular, the displacement numbers $n$ in the above constructs are constant positive numbers. For templating, we refine the constructs to support template expressions in place of positive constants. One of the consequences of adding template variables is that the semantics of contracts now depends also on mappings of template variables in a *template environment* $\mathsf{TEnv} : \mathsf{Var} \to \mathbb{N}$, which is also the case for many temporal properties of contracts. For example, the type system for ensuring causality of contracts [3] and the concept of horizon are now parameterized by template environments.

## 3 The Payoff Intermediate Language

The main motivation behind the payoff language is to bridge the gap between the contract DSL and a traditional expression language, which is usually used to implement pricing engines. The

payoff language should be relatively straightforward to compile to various target languages such as Haskell, Futhark [5], or OpenCL.

$$il ::= \texttt{now} \mid \texttt{model}(l,t) \mid \texttt{if}(il,il,il) \mid \texttt{loopif}(il,il,il,t) \mid \texttt{payoff}(t,p,p) \mid unop(il) \mid binop(il,il)$$

$$unop ::= \texttt{neg} \mid \texttt{not} \qquad binop ::= \texttt{add} \mid \texttt{mult} \mid \texttt{sub} \mid \texttt{lt} \mid \texttt{and} \mid \texttt{or} \mid \ldots \qquad t ::= n \mid i \mid v \mid \texttt{tplus}(t,t)$$

The payoff language is an expression language ($il \in \mathsf{ILExpr}$) with binary and unary operations, extended with conditionals and generalized conditionals `loopif`, behaving similarly to `checkWithin`. Template expressions ($t \in \mathsf{TExprZ}$) in this language are extensions of the template expressions of the contract language with integer literals and addition. Terms in the payoff language can be evaluated given a proper external environment, a proper template environment, and a *discount function* $d : \mathbb{N} \to \mathbb{R}$. The result of the evaluation is a single real value in contrast to the contract language for which the semantics is given in terms of traces.

## 4 Compiling Contracts to Payoffs

The contract language consist of two levels, namely constructors to build contracts ($c$) and expressions used in some of these constructors (`scale`, `checkWithin`, etc.). We compile both levels into a single payoff language. The compilation functions $\tau_e [\![-]\!] : \mathsf{Expr} \times \mathsf{TExprZ} \rightharpoonup \mathsf{ILExpr}$ and $\tau_c [\![-]\!] : \mathsf{Contr} \times \mathsf{TExprZ} \rightharpoonup \mathsf{ILExpr}$ are recursively defined on the syntax of expressions and contracts, respectively, taking the starting time $t_0 \in \mathsf{TExprZ}$ as a parameter.

$$\tau_e [\![\texttt{cond}(b,\ e_0,\ e_1])]\!]_{t_0} = \texttt{if}(\tau_e [\![b]\!]_{t_0}, \tau_e [\![e_0]\!]_{t_0}, \tau_e [\![e_1]\!]_{t_0}) \qquad \tau_c [\![\texttt{zero}]\!]_{t_0} = 0$$

$$\tau_e [\![\texttt{obs}(l,i)]\!]_{t_0} = \texttt{model}(l, \texttt{tplus}(t_0, i)) \qquad \tau_c [\![\texttt{translate}(t,c)]\!]_{t_0} = \tau_c [\![c]\!]_{\texttt{tplus}(t_0,t)}$$

$$\tau_c [\![\texttt{transfer}(p_1,p_2)]\!]_{t_0} = \texttt{payoff}(t_0,p_1,p_2) \qquad \tau_c [\![\texttt{both}(c_0,c_1)]\!]_{t_0} = \texttt{add}(\tau_c [\![c_0]\!]_{t_0}, \tau_c [\![c_1]\!]_{t_0})$$

$$\tau_c [\![\texttt{scale}(e,c)]\!]_{t_0} = \texttt{mult}(\tau_e [\![e]\!]_{t_0}, \tau_c [\![c]\!]_{t_0})$$

$$\tau_c [\![\texttt{checkWithin}(e,t,c_1,c_2)]\!]_{t_0} = \texttt{loopif}(\tau_e [\![e]\!]_{t_0}, \tau_c [\![c_0]\!]_{t_0}, \tau_c [\![c_1]\!]_{t_0}, t)$$

Let $\mathcal{E} [\![e]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \rightharpoonup \mathbb{R} + \mathbb{B}$, $\mathcal{C} [\![c]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \rightharpoonup \mathbb{N} \to \mathsf{Party} \times \mathsf{Party} \to \mathbb{R}$, and $\mathcal{IL} [\![il]\!] : \mathsf{ExtEnv} \times \mathsf{TEnv} \times (\mathbb{N} \to \mathbb{R}) \times \mathsf{Party} \times \mathsf{Party} \rightharpoonup \mathbb{R} + \mathbb{B}$ define the semantics of the contract expression sublanguage, the semantics of contracts, and the semantics of the payoff language, respectively. We also assume a function $HOR : \mathsf{Contr} \times \mathsf{TEnv} \to \mathbb{N}$ that returns a conservative upper bound on the length of a contract. The compilation function satisfies the following properties:

**Theorem 1** (Soundness)**.** *Assume parties $p_1$ and $p_2$ and discount function $d : \mathbb{N} \to \mathbb{R}$.*

- *If $\tau_e [\![e]\!]_0 = il$ and $\mathcal{E} [\![c]\!]_{\rho,\delta} = v_1$ and $\mathcal{IL} [\![il]\!]_{\rho,\delta,d,p_1,p_2} = v_2$ then $v_1 = v_2$.*
- *If $\tau_c [\![c]\!]_0 = il$ and $\mathcal{C} [\![c]\!]_{\rho,\delta} = trace$, where $trace : \mathbb{N} \to \mathsf{Party} \times \mathsf{Party} \to \mathbb{R}$, and $\mathcal{IL} [\![il]\!]_{\rho,\delta,d,p_1,p_2} = v$ then $\sum_{t=0}^{HOR(c,\delta)} d(t) \times trace(t)(p_1,p_2) = v$.*

To avoid recompilation of a contract when time moves forward, we define a function $\mathsf{cutPayoff}()$. This function is defined recursively on the syntax of intermediate language expressions. The only interesting case is the case for `payoff` expressions:

$$\mathsf{cutPayoff}(\texttt{payoff}(t,p_1,p_2)) = \texttt{if}(\texttt{lt}(t,\texttt{now}), 0, \texttt{payoff}(t,p_1,p_2))$$

The function guards the `payoff` expression with a condition guarding whether this payoff should have effect. For the remaining cases, the function recurses on subexpressions and returns otherwise unmodified expressions.

Avoiding recompilation can significantly improve performance especially on GPGPU devices. On the other hand, additional conditionals are introduced, which results in a number of additional checks at runtime. Experiments conducted with "hand-compiled" OpenCL code, which

was semantically equivalent to the payoff language code, show that for the simple contracts, like European options, additional conditions, introduced by cutPayoff() do not significantly influence performance. The estimated overhead was around 2.5 percent, while compilation time is in the order of a magnitude bigger than the total execution time.

## 5 Conclusion

This work extends the certified contract management system of [3] with template expressions, which allows for drastic performance improvements and reusability in terms of the concept of instruments. Along with changes to the contract language, we developed a formalization of the payoff intermediate language in Coq. Our approach introduces the abstract syntax of the payoff language as an inductive data type with the semantics of the payoff language and the compilation from the contract DSL defined as partial functions (using Coq's Option data type).

A number of important properties (including soundness) of the translation from contracts to the payoff language have been proven in Coq. Moreover, Coq's code extraction mechanism is used to obtain a certified compiler implementation in the Haskell programming language. We are currently working on establishing the important property of the new time-parameterized payoff evaluation function, which states that given a contract $c$ and its compilation into payoff code $il$, evaluating $il$ at time $t$ gives us the same result (upto discounting) as first advancing $c$ to time $t$, then compiling it to payoff code, and then evaluating the result at time 0.

## References

[1] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer*, 8(6):485–516, 2006.

[2] B.R.T Arnold, A. Van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, 1995.

[3] Patrick Bahr, Jost Berthold, and Martin Elsman. Certified symbolic management of financial multi-party contracts. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP'2015, pages 315–327, September 2015.

[4] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(1):27–45, 2009.

[5] Troels Henriksen, Martin Elsman, and Cosmin E Oancea. Size slicing: a hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 31–42. ACM, 2014.

[6] Tom Hvitved, Felix Klaedtke, and Eugen Zalinescu. A trace-based model for multiparty contracts. *The Journal of Logic and Algebraic Programming*, 81(2):72–98, 2012.

[7] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[8] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'2000, September 2000.

[9] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2015. Homestead revision, Founder, Ethereum & Ethcore, gavin@ethcore.io.

# A Secrecy-Preserving Language for Programming of Distributed and Object-Oriented Actor Systems *

Toktam Ramezanifarkhani and Olaf Owe

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

## 1 Introduction

Programming languages can provide fine-grained control for security issues because they allow accurate and flexible security information analysis of program components [4]. In particular, to specify and enforce information-flow policies, the effectiveness of language-based techniques has been established. Information-flow policies are essentially specified based on a mapping from the set of logical information holders to a lattice of security classes representing levels of information sensitivity. Moreover, these policies usually dictate that no execution of the program should lead to an information-flow from more to less sensitive information holders [3].

Violations of information-flow policies cause illegal explicit or implicit information flows. Illegal explicit flows occur for instance when confidential (private or high) information is assigned to variables considered public or low. Moreover, giving a precise definition of the policy and thus illegal flows are challenging and highly depends on the model of the system, the assumptions about attackers and their power. In the literature, secure information flows are expressed by semantic models of program execution in the form of a *noninterference* policy. Noninterference stipulate that manipulation and modification of private data should be allowed in programs, as long as their visible outputs do not improperly reveal information about the private data. In addition, the attackers are assumed to be able to view "low" information. The usual method for showing that noninterference holds is to demonstrate that the attacker cannot observe any difference between two executions that differ only in their confidential input [2]. In other words, if two possible input states of a program share the same low values, then the observable behaviors of the program execution on these states should be indistinguishable by the attacker [4]. In general the observable behavior is defined by the program output. However, there is no limitation in specifying program behaviors, and there is not a fixed limitation of the kinds of (explicit or implicit) program behavior observable by the attackers.

Although indistinguishablility of the program output by the attackers is a necessary condition to satisfy noninterference, we believe that the traditional notions of obervability are not enough because sometimes the set or the sequence of method calls are informative to the attackers and may reveal high level information even if the (traditional) noninterference between the program input and output is satisfied. For example, in the following statement

```
if private-var=1 then x:=o.m(e) else skip,
```

where the syntax $x := o.m(e)$ denotes a remote method call to $o$, an attacker may deduce information about the private variable based on the observation of method calls. This is a case of indirect information flow, which may appear when the observable program behavior includes observation of method calls. In such cases, in addition to leakage of the program private

*This work was done in the context of the *IoTSec* project funded by the Norwegian Research Council (http://cwi.unik.no/wiki/IoTSec:Home).

information, by sniffing the network traffic or monitoring library function and system calls, an attacker can observe privilege raise and drop activities related to the system resources [1]. Therefore, we introduce a notion of *Interaction Noninterference*, which stipulates program executions to be equivalent in the view of attackers observing method calls. Violation of interaction noninterference is critical in distributed systems with message passing.

### Application Platform: Creol

Creol [5] is a language for programming of concurrent object-oriented systems that exploits the paradigm of active objects using asynchronous method calls as the only interaction mechanism, thereby combining the Actor model and object-orientation. Shared variables as well as thread-based notification are avoided. A local suspension mechanism allows an object to perform other tasks while waiting for a condition to become true or for a method result to appear. In Creol an object is like a black box in the scene that its contents such as its fields are not observable from the outside of the object, and the main observation of objects is their interaction via method calls. In a network this is observed through messages corresponding to initiation and completion of remote method calls. Underlying network protocols may ensure that an attacker may observe but not alter the content of a message [3]. And message content may be considered non-observable due to encryption techniques. However, the destination and source of a message is considered observable since these often may be explicitly or implicitly deduced. In addition, confidential message content communicated through the network to untrusted objects is also a source of the secrecy leakage when received by such objects. Our approach covers these kinds of information leakage, and is also relevant in Actor-based systems. The notion of observable events may be further refined, for instance by considering certain parts of the network secure, say locally created objects.

Creol is interesting with respect to security analysis because it captures essential mechanisms of distributed systems, and at the same time includes elements that would normally belong to of the operating system, such as the underlying process queue. It is therefore possible to build a virtual machine for Creol with a simple and well-protected interface, thereby avoiding some low-level security issues. The simplicity of its formal semantics is a good basis for formal analysis, and for extensions of the language with build-in support of security notions.

We consider a minimal core language inspired by Creol, and extend it to consider sensitivity of information in the so-called *Secrecy Preserving Creol* (SeCreol) language. Furthermore, the Creol encapsulation mechanisms allow us to consider a dynamic type system for security levels, integrated in the operational semantics, which can be approximated by the static typing system.

### Language Extension: SeCreol

SeCreol is a language for programming of concurrent object-oriented systems that maintains security information as well as type information, such that considering the noninterference policy, typable programs are provably secure. Security information in SeCreol are assigned to each object and program element stating who (i.e., which objects in which security classes) can have direct or indirect read access to them, denoted $(r, ir)$. Therefore, to have a system that preserves secrecy of information, we use a combination of access control and information flow by determining the security class for each object capturing direct access, and tracking security information through computation paths to capture indirect accesses. Moreover, access such as direct and indirect read can be interpreted different for different program elements. For example, in case of a method, this security information determines the security class of objects that can call the method, and can be affected by its execution.

$$\begin{array}{llll} \xi & ::= & (r, ir) & \text{C-property} \\ \psi & ::= & (T, \xi) & \text{TC-property} \\ S & ::= & \psi\ m([\psi\ x]^*) & \text{method signature} \end{array}$$

$$\text{(T-CALL)}\quad \frac{\Gamma_c \vdash x : \psi \quad \Gamma_c \vdash m_\psi : (\psi' \mapsto \psi") \qquad \Gamma_c, L_c \vdash e : \tau \quad \Gamma_c \vdash \tau \leq \psi' \quad \Gamma_c \vdash \psi \geq \psi'}{\Gamma_c, L_c \vdash x.m(e) : \psi"}$$

Figure 1: A Part of SeCreol language syntax and secure-type system.

To preserve secrecy, i.e., data confidentiality and noninterference, and thus to prevent information leakage, we built on the BLP model as the security model [6]. An example of a part of the syntax and security-type system of SeCreol is given in Fig. 1. where $T$ denotes a type, the *C-property* captures secrecy, and the *TC-property* ($\psi$) captures their combination. Here $\Gamma_c$ is the TC typing environment and $L_c$ captures the security level of (any) enclosing branch conditions. The example type rule states that in a context with security class $L_c$, objects can call a method as long as the method does not access a higher class value as its input. Apart from this, the type system must track call traces in different branches of an if-statement with a high security context and compare them at the branching end. The secure-type system satisfies preservation and progress. Moreover, we present the operational semantics to explicitly check for secrecy errors during reduction. Finally, we prove that our secure-type system is sound which ensures that every well-typed program of our language satisfies the proposed noninterference property.

### Conclusion

Our main contributions are: (i) Introducing interaction noninterference as a more permissive property addressing an implicit channel. (ii) Extending the Creol language by providing a security-type system and operational semantics to preserve secrecy. (iii) Provably enforce the interaction noninterference property in programs of the SeCreol language.

There is evidence, for example in [1], that shows that the interaction between components and objects, e.g., the sequence of system and method calls, are valuable for attackers and can cause information leakage. Therefore, interaction noninterference is a critical property in a variety of secure systems, and thus its satisfaction and application is not limited to object-oriented systems.

## References

[1] S. Christey, J. E. Kenderdine, and et.al. Common weakness enumeration (cwe version 2.9), 2015.

[2] Joseph A Goguen and José Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, pages 75–75, 1984.

[3] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*. IOS Press, 2012.

[4] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *POPL'98*, POPL'98, pages 365–377. ACM, 1998.

[5] Einar Broch Johnsen and Olaf Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, March 2007.

[6] John McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, 1990.

# Don't let data *Go* astray

## A Context-Sensitive Taint Analysis for Concurrent Programs in Go

Ka I Pun[1], Martin Steffen[1], Volker Stolz[1,2], Anna-Katharina Wickert[3], Eric Bodden[4,5], and Michael Eichberg[3]

[1] University of Oslo, Norway
[2] Bergen University College, Norway
[3] Technische Universität Darmstadt, Germany
[4] Paderborn University, Germany
[5] Fraunhofer IEM, Germany

**Abstract**

*Taint analysis* is a form of data flow analysis aiming at secure information flow. For example, unchecked user input is considered typically as "tainted", i.e., as untrusted and potentially dangerous. Untrusted data may lead to corrupt memory, undermine the correct functioning or privacy concerns of the software otherwise, if it reaches program points it is not supposed to. Many common attack vectors exploit vulnerabilities based on unchecked data and the programmer's negligence of foreseeing all possible user inputs (including malicious ones) and the resulting information flows through the program.

We present a static taint analysis for Go, a modern, statically typed programming language. Go in particular features concurrent programming, supporting light-weight threads dubbed "goroutines", and message-based communication. Beside a classical context-sensitive taint analysis, the paper presents a solution for analyzing channel communication in Go. A longer version of the material will appear in [2].

## 1 Motivation

The high-level programming language Go is gaining traction, being used in various software products like Docker and Dropbox, as well as for Android and iOS applications [4]. Support for concurrency based on goroutines and channel communication lies at the very core of Go's design. While standing, at least syntactically and in spirit, in the tradition of C, Go is lightyears ahead compared to the more low-level language C, when it comes to considering "secure" computation. Two notable improvements on that front are Go's advanced static type system and the absence of pointer arithmetic. But still, the concurrency features and the intended distributed and mobile applications and platforms, make the task to ensure secure computation more challenging. In open, interactive and even mobile applications, it is even more vital: One needs to ensure that unchecked, arbitrary user input does not reach sensitive points in the program, potentially leading to memory corruption or other violations, and that private data does not go astray, leaking to the outside or other applications.

To counter such vulnerabilities, we have developed and implemented a static taint analysis, which is a specific form of flow analysis, for Go. The analysis is context-sensitive, covers reference types and relies on the corresponding packages of the Go-compiler. In particular, we cover sound analysis of taint information flow for channel-based communication.

# 2 Static taint analysis

Information flow analysis [3] is widely studied and used for many languages, including Java and C, but is currently not implemented for Go. A taint analysis identifies flows of private information to untrusted places, e.g., a SMS to the attacker, or untrusted information such as user input passed on or processed without sanity checks. We will concentrate on the former type (privacy-based) of taint analysis. The analysis defines private information and untrusted places through a list of *sources* and *sinks*. In that it can be seen as a form of a *def-use*-analysis. Sources are API calls which could read sensitive data, and sinks are API calls which can write data to an untrusted place. Our taint analysis is static and context-sensitive, concentrating on direct information flows.

## 2.1 The Go programming language

The small example from Listing 1 illustrates the analysis, highlighting a few Go features. The main function spawns a call to $f$ as a new goroutine (think "thread") using the keyword `go`, handing over the freshly created synchronous channel via the actual parameter `ch`. The channel is (obviously) shared between parent and child goroutine, and used to communicate string values from parent to child and, thereby, also synchronizes between them: sending to the channel is done in location $n_6$ and reading from it in location $f_2$. The example defines two functions *source* and *sink*, whose only purpose here is to illustrate the concepts of sources and sinks for the taint analysis. Go supports pointers (though no pointer arithmetic as in C): In the example, the argument to the *sink*-function is passed as a reference.

```
    func main() {
n1      x := "hello, world"
n2      ch := make(chan string)
n3      go f(ch)
n4      sink(&x)
n5      x = source()
n6      ch ← x
    }
f1  func f(ch_1 chan string) {
f2      y := ←ch_1
f3      sink(&y)
    }
    func sink(s *string) {}
    func source() string {
        return "secret"
    }
```

Listing 1: A Go program with channels, goroutines, pointers.

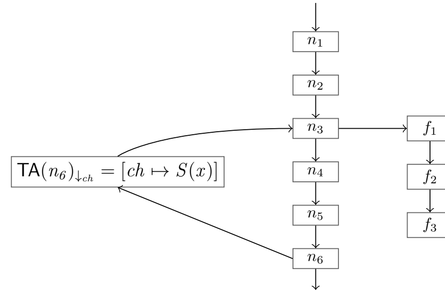



Figure 1: Flow graph for Listing 1 with the additional edge for a write to a channel. Edges to sinks/from sources elided for simplicity.

## 2.2 Implementation

Building upon similar analyses such as [5], the implementation is context sensitive, i.e., a function call (including *asynchronous* functions) is analyzed differently depending on the call-site *context*, which, in our case, is a "value context" containing information about the taint status of the actual parameters. The analysis partly relies on existing technologies and libraries.

First, we use Go's static single-assignment (SSA) intermediate representation for a worklist-based iteration algorithm. Secondly, we use an inclusion-based pointer analysis (see [1]) to handle pointers in a more precise way.

Figure 1 shows the flow graph for the example program, where the nodes correspond to the "line annotations" in the code. The implementation uses a worklist algorithm for fixpoint iteration, working through the nodes. In the example, the *sink*-function is called twice, once in the main function with the untainted string value `"hello, world"`, and in the child goroutine executing $f$. In the latter case, the value delivered to the sink is received via the channel connecting the two goroutines, and in that case the value originates from the *source*-procedure. To correctly have the analysis report a possible flow of tainted data from a source to a sink, we introduce for each created channel a new type of data flow node, with incoming edges from the send-operations to the channel and outgoing edges to read-operations that may access the channel. The transfer function for the new node only transfers the lattice value of the channel (see the node labelled TA(...) in Figure 1), and not the complete lattice of the writing node.

Listing 1 uses the unbuffered channel only once. In larger programs buffered channels or many writes to a channel are likely. If a channel receives a `tainted` and an `untainted` value, the static analysis will over-approximate the channel's taint status to the lattice's top value $\top$.

# 3 Potential for monitoring

For some paths, a static analysis overapproximates the results, and a dynamic analysis under-approximate the results. If we introduce monitoring for these parts, we can improve our results. A scenario is to use monitoring as a sanitizer. A sanitizer works like a white list and changes a tainted value to an untainted, e.g. applying the hash function on a password should change the lattice value to untainted. A classical taint analysis ignores that the hash function changes the taint status to untainted and produces a false positive. A more complicated approach is to monitor the concurrent behavior of a program. The idea is to influence the scheduling so that a tainted value will only take safe paths.

# References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, 1994. Available as DIKU report 94/19.

[2] E. Bodden, K. I. Pun, V. Stolz, M. Steffen, and A.-K. Wickert. Information flow analysis for Go. In *7th Intl. Symp. On Leveraging Applications of Formal Methods, Verification and Validation*, volume 9952 of *LNCS*. Springer Verlag, Oct. 2016.

[3] D. Denning. *Secure Information Flow in Computer Systems.* PhD thesis, Purdue University, 1976.

[4] Go Authors. Mobile golang/go Wiki GitHub, Feb. 2016. URL https://github.com/golang/go/wiki/Mobile.

[5] R. Padhye and U. P. Khedker. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *2nd ACM SIGPLAN Intl. Workshop on State Of the Art in Java Program Analysis*, SOAP '13, pages 31–36, New York, NY, USA, 2013. ACM.

# Integration of Runtime Verification into Metamodeling

F. Macías[1], T. Scheffel[2], M. Schmitz[2], R. Wang[1],
M. Leucker[2], A. Rutle[1], and V. Stolz[1]

[1] Bergen University College, Norway
{fernando.macias, rui.wang, volker.stolz, adrian.rutle}@hib.no
[2] Institute for Software Engineering and Programming Languages,
University of Lübeck, Germany
{scheffel, schmitz, leucker}@isp.uni-luebeck.de

## 1 Domain Specific Modeling Languages (DSMLs)

Modeling is a well-established practice in the development of big and complex software systems. Domain Specific Modeling Languages (DSMLs) are a technique used for specifying such systems in an abstract way. These languages define the structure, semantics and constraints for models related to the same application domain. The models created with DSMLs are then interconnected or related to one another. Among the reasons for tailoring a language to the problem space is their better understandability by domain experts, capacity for high-level abstraction, and user friendliness. However, the use of DSMLs (like the use of types in general) does not shield the produced software from bugs or man-made mistakes. Software failures may still occur on complex systems due to a variety of reasons such as design errors, hardware breakdown or network problems. Ruling out these failures requires that verification methods that guarantee correct execution even in corner cases are integrated into the development process.

We previously presented our ideas for integrating specifications tighter with the model [4]. We improve on our realisation in the context of behavioural models for embedded systems [6], and now tackle modelling and specification of a distributed system. LEGO MindStorm® robots serve as the platform for a small case study.

## 2 Runtime Verification (RV)

The use of verification methods during the specification of a system can greatly improve their reliability. A commonly used verification technique is testing. Unfortunately, testing is seldom exhaustive and cannot always guarantee correctness. An exhaustive option to check every execution path is model checking. But this alternative may suffer the state space explosion problem [3], especially relevant in distributed systems due to their inherent non-determinism. Yet another possibility in the system verification domain is to use runtime verification (RV). Runtime verification is an approach growing in popularity to verify the correctness of complex and distributed systems by monitoring their executions. It can cope with the inadequacies of testing by reacting to systems' failures as soon as they occur. Also, it is a much more lightweight technique when compared to model checking, since only one execution path is checked. Generally, RV can be used to check whether the current execution of the complex system violates given correctness properties [3]. Such checking can be typically performed and decided by using a monitor which, in the simplest form, outputs either true or false.

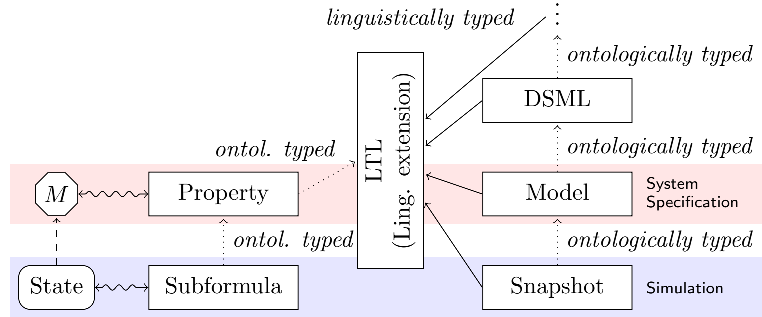



Figure 1: Multilevel model hierarchy with linguistic extension.

## 3 Combining RV and DSMLs

Our research idea is to integrate RV and domain specific modeling into the development process of complex systems. Such integration is achieved by linking the elements of the system model with the atomic propositions of the temporal correctness properties used to specify monitors. In order to achieve this idea, we view a system as having a state consisting of a set of atomic propositions. In RV, we specify correctness properties based on the atomic propositions and generate monitors from them. With this, monitor statements about the correctness of the current execution of the system can then be made. To be compatible with this view of a system, we define a multilevel modeling hierarchy [1] where the DSML and the actual model of the system are included. Moreover, the hierarchy includes instances of the system model that represent the particular state of the system at a given point in time. We call the instances of the system model *snapshots*. A snapshot is also a model, and contains the set of active elements of the system. The way in which the system evolves during the simulation is described using model transformations that generate a new snapshot from the previous one. In short, to link both RV and DSMLs, we associate the atomic propositions, used in RV, with the current state of the system, represented as a snapshot through execution or simulation of a domain specific model. Fig. 1 shows the different modeling levels and their elements in relation to each other. From a metamodeling perspective, every model on the right-hand side of the figure is *ontologically typed* by the model on top of it. The relation with the linguistic extension in the center of the picture is established by *linguistic typing*, that allows to give a second type to every element, orthogonal to the previous one [7]. Our implementation of a multilevel metamodelling tool that supports these concepts is described in [5] and available from http://prosjekt.hib.no/ict/multecore/.

### 3.1 Linking a DSML with temporal properties

Additionally, together with an example DSML, we also define the integration of its behavioral semantics with the evaluation of temporal properties. We achieve such integration by linking the elements of the DSML with the atomic propositions used in Linear Temporal Logic (LTL) formulas. We implement the syntax of a temporal logic as a linguistic extension. The key concept of a linguistic extension for our work is the possibility to connect *any* type of element or set of elements in the modeling hierarchy to the model representing the temporal property. We connect single elements in the snapshots to the temporal properties: an atomic proposition

in the LTL property is a fragment of a model instance that may appear in a snapshot. The atomic propositions are evaluated as follows: if at least one match of the fragment appears in the current snapshot of the system, the atomic proposition is evaluated to true, otherwise to false. This connection of elements and atomic propositions allows us to look at the sequence of snapshots of a system as the system's execution, from both RV and modelling points of view. So we can do RV with temporal logics in a natural way based on those snapshots, because they represent the states of the system during the execution. By doing so the connection between the correctness property and the modeled behavior of the system is kept consistent throughout the software engineering process, as propositions always refer to existing model elements.

### 3.2 Extensions

To make our approach more widely applicable, we present a detailed metamodel that captures a wider range of aspects of the robots (sensors, motors, communication), and allows design of distributed systems. Distribution can be modeled through replication of existing instances on the modeling level, or through composition. Upon deployment, code generated for a robot in the model is mapped to the corresponding physical entity.

As the property specification language spans the entire model, this gives naturally rise to partial monitors in each system that contribute to a global property. Annotations structure the evaluation of the sub-formulas into a hierarchy with an explicit mapping to individual instances. Code generation for the runtime monitors takes this into account and yields partial monitors using the ideas of Distributed Temporal Logic (DTL, [8]).

The connection of the atomic propositions to the model elements (states, sensors and motors) enables us to expand our approach to different LTL semantics like $LTL_3$ [2] in order to report final fulfillment or violation of the correctness properties as soon as possible when monitoring. Timed LTL would allow us to express real-time properties, and we can add other theories for using variables instead of only boolean propositions.

## References

[1] Colin Atkinson and Thomas Kühne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.

[2] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime Verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[3] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[4] Fernando Macías, Adrian Rutle, and Volker Stolz. A Property Specification Language for Runtime Verification of Executable Models. In *Nordic Workshop on Programming Theory (NWPT)*, pages 97–99, 2015. Tech. Rep. RUTR-SCS16001, School of Computer Science, Reykjavik University.

[5] Fernando Macías, Adrian Rutle, and Volker Stolz. MultEcore: Combining the best of fixed-level and multilevel metamodelling. In *3rd Intl. Workshop on Multi-Level Modelling*, CEUR Workshop Proceedings. CEUR-WS.org, 2016. To appear.

[6] Fernando Macías, Torben Scheffel, Malte Schmitz, and Rui Wang. Integration of runtime verification into meta-modeling for simulation and code generation. In *Intl. Conf. on Runtime Verification (RV'16)*, volume 10012 of *LNCS*. Springer, 2016.

[7] Alessandro Rossini, Juan de Lara, Esther Guerra, Adrian Rutle, and Uwe Wolter. A formalisation of deep metamodelling. *Formal Aspects of Computing*, 26(6):1115–1152, 2014.

[8] Torben Scheffel and Malte Schmitz. Three-valued asynchronous distributed runtime verification. In *Formal Methods and Models for Codesign, MEMOCODE*, pages 52–61. IEEE, 2014.

# On Formalizing Information-Flow Control Libraries*

Marco Vassena and Alejanro Russo

Chalmers University of Technology, Gothenburg, Sweden. {vassena,russo}@chalmers.se

Nowadays, people use apps found in app stores and software downloaded from the Internet for a variety of different tasks. Such apps usually manipulate users' private data and users, who wish to benefit from their functionality, are forced to grant them access to it. Unfortunately, users are often naive when it comes to security and trust apps that could have written by *anyone* with their private data, assuming that it will be actually used for the functionality advertised. Once apps collect users' information, there are no guarantees about what they will do with it, thus leaving room for data theft and data breach by malicious apps. The key to guaranteeing security without sacrificing functionality is not granting or denying access to sensitive data, but rather ensuring that data flows into appropriate places.

Information-Flow Control (IFC) [13] is a promising programming language-based approach to enforcing security. IFC scrutinizes how data of different sensitivity levels (e.g., public or private) flows within a program, and raises alarms when there is an unsafe flow of information. While most IFC tools require the design of new compilers, e.g., [8, 9, 10, 1], the functional programming language Haskell plays a unique privileged role: *it is able to enforce security via libraries* [6] by using an embedded domain-specific language. As long as developers program against the libraries API, the code is secure by construction. Different from other programming languages, Haskell type-system separates side-effect free from side-effectful computations—an essential feature to identify and avoid harmful side-effects capable to leak sensitive data. In recent years, researchers have increased their interest in such libraries, developing solutions that enforce IFC statically [7, 17, 12, 11], dynamically [15, 14], and as a combination of both [3].

Many IFC libraries, e.g., [7, 12, 15, 18], prove non-interference results by using the *term erasure* technique. It establishes a *simulation* between the evaluation of a program and its erased counterpart: a program does not leak secrets if it produces the same observable behavior regardless of the fact that secrets are erased *before* or *after* execution. Such proofs frequently involve to account for subtle interplay between programming languages features like, for instance, sub-computations and exceptions [16, 5], concurrency [2], and applicative functors [18]. It is precisely their complexity and their elusive interaction which make mechanized proofs, not only desirable, but needed to corroborate any security guarantee. To the best of our knowledge, there are no mechanized proofs of IFC libraries except for the core calculus of **LIO** [16] where no side-effectful operations are considered.

This ongoing work presents a full formalization of **MAC** [11] mechanically verified in Agda[1]. We formalize the core of **MAC** as a simply typed $\lambda$-calculus extended with booleans, unit values and monadic operations. The relation $t_1 \rightsquigarrow t_2$ represents the call-by-name small-step semantics of the calculus, which denotes that term $t_1$ reduces to term $t_2$. We write $\varepsilon_{\ell_A}\ (t)$ for the term obtained by erasing term $t$, that is by rewriting every piece of data above the attacker's security level, denoted by label $\ell_A$, to the special syntax node $\bullet$. The core of the proof technique consists in proving the *simulation* between term reduction and erasure function—the diagram in Figure 1 highlights this intuition. It shows that erasing sensitive data from a term $t$ and then taking a step (orange path) is the same as firstly taking a step and then erasing sensitive data (cyan path), i.e., the diagram *commutes*. If term $t$ leaks data whose sensitivity label is above $\ell_A$, then

*Based on On Formalizing Information-Flow Control Libraries in Proc. of ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, October 2016.

[1]Available at https://bitbucket.org/MarcoVassena/mac-agda

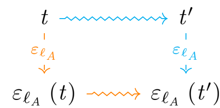



Figure 1: Single-step simulation.

erasing all sensitive data first and then taking a step might not be the same as taking a step and then erasing secret values—the leaked sensitive data in $t'$ might remain in $\varepsilon_{\ell_A}(t')$ after all.

We use this technique to show progress-insensitive and progress-sensitive non-interference for sequential and concurrent programs respectively. Furthermore, we discuss some interesting insights gained from the formalization, which also pertain to other IFC libraries like **LIO** and **HLIO**. Our results not only help identifying problems in existing IFC libraries, but also to assist library designers to faultlessly apply *term erasure* as a proof technique. More concretely, our mechanized proofs provide us with the following insights:

- **Context-aware erasure.** The decision of erasing terms might be context-dependent. For instance, erasing an argument in a multi-argument function might depend on the value (or type) of some other arguments. This dependency obstructs the definition of a sound *homomorphic* erasure function, complicating the analysis of security guarantees. We propose a novel *two-steps* erasure technique to repair such cases.

- **Masking sensitive exceptions.** In previous work, labeled exceptions are erased by erasing their content according to the sensitivity of their label, but always preserving their exceptional state [16]. In contrast, we propose to *mask* sensitive exceptions in erased programs by rewriting them to unexceptional values. While the simulation between terms and their erased counterparts guarantees that this rewriting is *sound*, the absence of exceptions in erased programs simplifies the verification of security properties.

- **Scheduler requirements.** When considering concurrent programs, we obtain a security proof which is valid for a wide-range of deterministic schedulers. We formally pin down sufficient requirements on the scheduler to guarantee progress-sensitive non-interference—a novel aspect if compared with previous work [14, 4]. As an example, we instantiate our results with a round-robin scheduler (the scheduler used by Haskell runtime system).

To the best of our knowledge, this is the first work of its kind for IFC libraries in Haskell, both for completeness and number of features included in the model. It is our hope that the insights gained by this ongoing work will help to formally verify other IFC programming languages.

## References

[1] N. Broberg, B. van Delft, and D. Sands. Paragon for practical programming with information-flow control. In *APLAS*, volume 8301 of *Lecture Notes in Computer Science*, pages 217–232. Springer, 2013.

[2] P. Buiras, D. Stefan, and A. Russo. On dynamic flow-sensitive floating-label systems. In *Proc. of the IEEE Computer Security Foundations Symposium*, CSF '14. IEEE Computer Society, 2014.

[3] P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. ACM, 2015.

[4] S. Heule, D. Stefan, E. Z. Yang, J. C. Mitchell, and A. Russo. IFC inside: Retrofitting languages with dynamic information flow control. In *Conference on Principles of Security and Trust (POST)*. Springer, April 2015.

[5] C. Hritcu, M. Greenberg, B. Karel, B. C. Peirce, and G. Morrisett. All your IFC exception are belong to us. In *Proc. of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2013.

[6] P. Li and S. Zdancewic. Encoding information flow in Haskell. In *Proc. of the IEEE Workshop on Computer Security Foundations*. IEEE Computer Society, 2006.

[7] P. Li and S. Zdancewic. Arrows for secure information flow. *Theoretical Computer Science*, 411(19):1974–1994, 2010.

[8] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, January 1999.

[9] F. Pottier and V. Simonet. Information Flow Inference for ML. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 319–330, January 2002.

[10] I. Roy, D. E. Porter, M. D. Bond, K. S. McKinley, and E. Witchel. Laminar: Practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. ACM, 2009.

[11] A. Russo. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015.

[12] A. Russo, K. Claessen, and J. Hughes. A library for light-weight information-flow security in Haskell. In *Proc. ACM SIGPLAN symposium on Haskell (HASKELL '08)*. ACM, September 2008.

[13] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[14] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, 2012.

[15] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, pages 95–106, New York, NY, USA, 2011. ACM.

[16] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in the presence of exceptions. *Arxiv preprint arXiv:1207.1457*, 2012.

[17] T. C. Tsai, A. Russo, and J. Hughes. A library for secure multi-threaded information flow in Haskell. In *Proc. IEEE Computer Security Foundations Symposium (CSF '07)*, July 2007.

[18] M. Vassena, P. Buiras, L. Waye, and A. Russo. Flexible manipulation of labeled values for information-flow control libraries. In *Proceedings of the 12th European Symposium On Research In Computer Security*. Springer, September 2016.

# Types for CAS: Relaxed Linearity with Ownership Transfer*

Elias Castegren and Tobias Wrigstad

Uppsala University
first.last@it.uu.se

**Abstract**

This extended abstract overviews work on a type system for lock-free programming based on compare-and-swap. The type system prevents atomicity violations in lock-free programs, where insertion and removal of objects from a linked structure would be subject to data-races breaking linearity of ownership. The type system has successfully been applied to a small number of lock-free data structures.

## 1 Introduction

Modern hardware is increasingly relying on multi-cores to achieve performance [2]. But with the power of parallelism comes the responsibility of synchronisation – two threads must not be allowed uncontrolled access to the same memory location, as this could lead to data-races and the problems that follow (*e.g.,* lost updates).

On one end of the spectrum we have dynamic synchronisation techniques like locks, where threads have mutually exclusive access to a piece of data, and non-blocking techniques like lock-free algorithms, where each thread follows some protocol in order to guarantee that two operations in conflict cannot both succeed [8]. On the other end of the spectrum we have static techniques like type systems specialised for parallel programming (*e.g.,* [1, 4, 6]).

A very powerful property is the concept of a *linear* (or *unique*) reference, which is a reference that is guaranteed to have no aliases. Like with locks, a thread accessing an object through a linear reference can assume exclusive access, but without the potential runtime overhead of blocking. This however comes at the cost of banning sharing altogether, and requires explicitly passing the reference between threads in order to transfer ownership. This restriction is too strong for many concurrent applications. For example, lock-free algorithms typically require concurrent updates to shared mutable state.

This paper sketches a recently developed type system that uses a relaxed notion of linearity that is powerful enough to guarantee mutual exclusion – a thread always has exclusive access to the linear resources of an object – but at the same time flexible enough to express lock-free data structures where several threads compete to assert ownership of such resources.

**Outline** § 2 presents the notion of relaxed linearity and ownership transfer and § 3 exemplifies the expressiveness of the system by showing the implementation a Treiber stack [9]. A full treatise of the type system together with more examples, a formalisation and proof of soundness and data-race freedom can be found in our technical report [3].

## 2 Relaxed Linearity

Traditionally, linear references are references that are statically guaranteed not to have any aliases. They trivially provide mutual exclusion, as holding a linear reference implies holding the *only* reference to an object. Another way to say this is that data-races are only possible if two threads have shared access to some data.

However, aliasing in itself does not necessarily lead to data-races. An important observation is that having several aliases to an object is safe from a data-race perspective as long as at most one of the aliases can be used to access the contents (*i.e.,* the fields) of that object. This is a strong notion of *ownership*, which implies permission to update the object.

*This work is sponsored by the UPMARC centre of excellence, the FP7 project “UPSCALE” and the project “Structured Aliasing” financed by the Swedish Research Council.

Whereas traditional linear types impose a uniqueness restriction on references, we allow unbounded aliasing but require that *ownership* is treated linearly. This allows threads to share objects arbitrarily as long as at most one of the aliases owns the object, and this ownership is never duplicated. Classic linearity equates transferring a reference with transfer of ownership, but we can also transfer ownership between existing aliases. This allows setting up aliasing and later getting the right ownership in place.

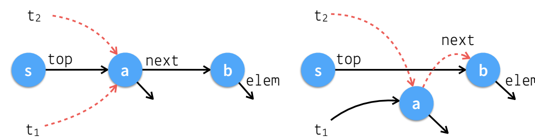



Figure 1: A Treiber stack before and after a successful pop. Black arrows show references with (linear) ownership. Red dashed arrows show non-owning references.

Figure 1 shows a data-structure before and after a transfer of ownership. Black arrows show references with (linear) ownership, and red dashed arrows show references without. Note that no object has more than one black arrow to it. In the left half, `top`, $t_1$ and $t_2$ are all aliases, but the latter two have no ownership. In the right half, the ownership in `top` has been transferred to $t_1$, and the ownership in `a.next` has been transferred to `top`, overwriting its old value. § 3 presents the implementation of the stack that Figure 1 is illustrating.

Further: not all concurrent access patterns to the same memory are harmful. Two threads reading the same memory is trivially safe, but so is having two threads racing to perform an atomic compare-and-swap (`CAS`) to the same memory location a in a (correct) lock-free algorithm. As long the risk of a potential data-race is explicated and its side-effects can be limited, it is safe to have one or more points of contention in a shared data structure.

In our type-driven approach, the programmer must mark the fields of an object which may be subject to (and statically allow) concurrent updates, without requiring that the writer first asserts ownership of the object. Since the value of such a field can change at any point in time, we call the reading of such a field a *speculation*. The field itself is marked by a **spec** keyword. In Figure 1, the `top` field is a **spec** field, meaning it can be updated by any thread

In order to protect the linearity of an object stored in a **spec** field, we require that reads and writes to this field are performed atomically. Specifically, reading a **spec** field creates an alias without ownership. In order to transfer ownership from a **spec** field, the old value must be atomically overwritten using an atomic `CAS` operation. Due to linear ownership, the overwritten field held the *only* ownership of its referenced object, and so a thread can assert ownership of that object after a successful `CAS`. Aliases without ownership can be used for the compare part of the compare-and-swap. In Figure 1, the value in `top` is overwritten, and so its ownership can be transferred into $t_1$ without duplicating ownership.

The next section introduces our system by example – the implementation of a lock-free stack.

## 3 A Lock-free Stack

Figure 2 shows the implementation of a Treiber stack [9] in a language using our type system. Each `Node` is linear and contains an element of some elided linear type `T` as well as a reference to the next node in the stack. The `elem` field is protected by linear ownership, meaning there can be at most one reference to a `Node` that has permission to access `elem`. Relaxed linearity however allows the existence of non-owning aliases of a `Node`, which lets threads perform the speculation necessary to perform atomic operations. The `next` field is immutable and can therefore be safely accessed concurrently. Immutable fields are also guaranteed to be stable, so that a value cannot change underfoot. The single point of contention, the `top` field in the stack head, is easily identifiable as it is the only **spec** field in the data structure.

The reads on Lines 13 and 23 give non-owning aliases of the top field of the stack (*cf.,* the red dashed arrows in Figure 1). When the `CAS` in the `pop` function succeeds, ownership is transferred from the `top` field to the variable `t`, which can then be used to destructively read the `elem` field on Line 15. Figure 1 shows how references move and exchange ownership during a successful `pop` operation. Note that $t_1$ and `s.top` are actually aliases, but the type system makes sure that no thread can extract additional ownership from `t.next` after the `CAS`, even though there might be other threads reading the field (*e.g.,* through $t_2$`.next`).

The write to the immutable `next` field on Line 24 is allowed since the object is not visible to other threads yet. As soon as the object is published by the `CAS`, the type system prevents further writes to the field. The resulting ownership transfer of the `CAS` operation in `push` can be understood by reading

Figure 1 from right to left, replacing $t_1$ by `n` and ignoring the reference $t_2$ (which cannot exist as the `Node` was created by the current thread and has not yet been shared).

```
1  struct Stack {
2    spec top : Node
3  }
4
5  struct Node {
6    var elem : T // var fields are protected
7                 // by ownership
8    val next : Node // val fields are "final"
9  }
10
11 def pop(s : Stack) : T {
12   while(true) {
13     val t = s.top;
14     if (CAS(s.top, t, t.next)) then
15       return consume t.elem;
16   }
17 }
18
19 def push(s : Stack, e : T) : void {
20   val n = new Node;
21   n.elem = consume e;
22   while(true) {
23     val t = s.top;
24     n.next = t;
25     if (CAS(s.top, n.next, n))
26       break;
27   }
28 }
```

Figure 2: A Treiber Stack

Note that the type system preserves linearity of ownership and guarantees that our implementation is data-race free – if a thread manages to pop a `Node`, no other thread can succeed in getting access to the same `Node`. There is no difference in runtime overhead when compared to other standard implementations of the Treiber stack.

# 4 Conclusions

This paper presented a brief overview of a relaxed notion of linearity that separates aliasing from ownership. Our technical report contains the details of the underlying type system, together with a formalisation of a simple imperative language using it, and proof of soundness and data-race freedom [3]. Apart from the Treiber Stack, we have used the language to implement a Michael-Scott Queue and a Tim Harris List. The type system is non-intrusive as most types can be inferred (Figure 2 compiles and runs in our prototype implementation as is, modulo minor syntactic variations).

Other more powerful verification techniques for concurrent data structures exist (*e.g.,* Separation Logic or Rely-Guarantee References [5]), but we are not aware of other *type systems* aimed at implementing lock-free algorithms. There are type systems that rely on linearity for atomic transfer of ownership (*e.g.,* Rust[7]), but we have not seen this done without using locks or destructive reads, which precludes lock-free implementations.

Our type system captures existing patterns of lock-free programming and enforces correct usage though typing discipline. While we cannot guarantee correctness of a lock-free implementation, with minimum overhead in programmer effort we can exclude data-races and any behavior leading to two threads believing that they own the same value.

# References

[1] R. Bocchino. An effect system and language for deterministic-by-default parallel programming, 2010. PhD thesis, University of Illinois at Urbana-Champaign.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[3] E. Castegren and T. Wrigstad. Lolcat: Relaxed linear references for lock-free programming. Technical Report 2016-013, 2016. Uppsala University.

[4] E. Castegren and T. Wrigstad. Reference Capabilities for Concurrency Control. In *ECOOP*, 2016.

[5] C. S. Gordon. *Verifying Concurrent Programs by Controlling Alias Interference*. PhD thesis, University of Washington, 2014.

[6] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *ACM SIGPLAN Notices*, volume 47, pages 21–40. ACM, 2012.

[7] N. D. Matsakis and F. S. Klock, II. The rust language. In *HILT*, 2014.

[8] M. Moir and N. Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[9] R. K. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

# Describing Behaviour Models through Reusable, Multilevel, Coupled Model Transformations

Adrian Rutle[1], Fernando Macias[1], Francisco Durán[2], Roberto Rodriguez-Echeverria[3], and Uwe Wolter[4]

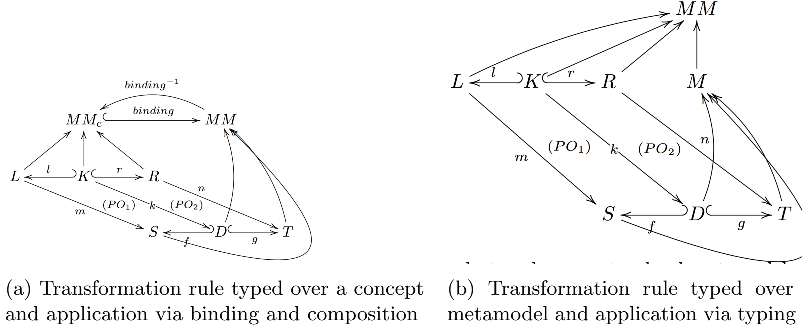
[1] Bergen University College, {aru,fmac}@hib.no
[2] University of Málaga, duran@lcc.uma.es
[3] University of Extremadura, rre@unex.es
[4] University of Bergen, uwe.wolter@uib.no

Model-driven software engineering (MDSE) is one of the emergent responses from the scientific and industrial communities to tackle the increasing complexity of software systems. MDSE utilises abstractions for modelling different aspects – behaviour and structure – of software systems, and treats models as first-class entities in all phases of software development. While modelling structure has advanced both in industry and academia due to mature tools and frameworks, modelling behaviour has still a long way to go especially because of the challenges related to the definition of their semantics. Several approaches have been proposed for the definition of behaviour models based on model transformations. Since most behaviour models have some commonality both in concepts and their semantics, reusing these model transformations across behaviour models would be a huge gain.

Unfortunately existing approaches which employ reusable model transformations for the definition of behaviour models focus on traditional two-level modelling hierarchies and their affiliated two-level model transformations (see [7] for a survey). Two-level modelling hierarchies have a limited number of abstraction levels and describe software domains in terms of models and their instances. Using these hierarchies for the definition of behaviour modelling languages usually leads to unnecessary complexity and synthetic type-instance relations; in addition, it suffers from a limited number of abstraction levels [3]. Modelling behaviour is inherently multilevel since we define a metamodel for the behaviour modelling language while the semantics is described two levels below its metamodel [4, 10]. This is because the behaviour is reflected in the running instances of the models which conform to the metamodel. Unfortunately multilevel aware model transformations [2, 1] are relatively new and are not yet proven suitable for reuse and definition of behaviour models.

Therefore our proposal is based on multilevel (to support the inherent multilevelness of behaviour models) coupled (to avoid proliferation of rules) model transformations. In this paper, we will shortly outline existing approaches to reusable model transformations and explain their advantages and drawbacks w.r.t. the definition of behaviour models. Then, we briefly explain multilevel aware model transformations and evaluate their applicability to reusable definition of behaviour. Also, we propose a multilevel modelling hierarchy for the definition of behaviour modelling languages which supports reusable, multilevel coupled model transformations for definition of the semantics of these languages. Finally, we adopt the contract-based model transformation testing approach presented in [9] to validate the model transformations.

**Two-level transformation approaches.** In its most general terms, rule-based transformations are given by a set of transformation rules in which each rule is defined as a left pattern and a right pattern. The "untyped" version of transformation rules is by definition generic since their application only depends on finding a match of the left pattern in the model. However, untyped rules are less usual (especially) in model transformations which are used to define




(a) Transformation rule typed over a concept and application via binding and composition

(b) Transformation rule typed over a top level metamodel and application via typing composition

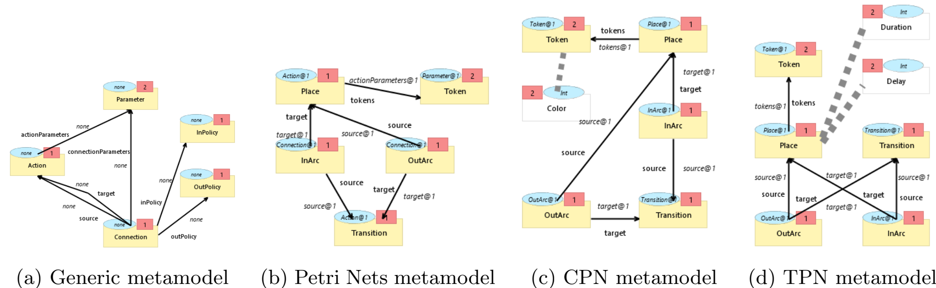
Figure 1: Approaches to reusable model transformations

behavioural models, since without the type information one cannot differentiate between the semantics of the different concepts in the modelling language. Take Petri nets as an example, it is necessary to treat places and transitions in different ways, since a generic untyped rule to define the semantics of these two concepts would not make sense.

In typed transformation rules, calculating the matches needs to fulfil a typing condition. Although useful in many cases, fixing the types of elements in the left and right patterns introduces restrictions when we reuse the rules on recurrent structure which has similar behaviour. To achieve genericness, in [11, 5] the rules are typed over a "generic" metamodel which is called *concept*. Then, any metamodel to which there exist an embedding from the concept-metamodel (called binding) can be used by composition for the type-check during matching (see Fig. 1a for the homogeneous version). Thus anywhere in a modelling hierarchy one can do typing to the concept metamodel and get the transformation rules for free. However, it is not always straight forward to define the embedding morphism from the concept metamodel $MM_c$ to the metamodel $MM$. This might be because the $MM$ has several structures which have the same behaviour leading to several bindings. The authors have solved this problem by introducing syntax for the definition of cardinality in which the concept metamodel can be written in a generic way, however, in the realisation of the concept, this is just syntactic sugar for the definition of multiple concept metamodels. Another challenge is related to finding a reasonable embedding due to structure mismatch (or heterogeneity). Adapters and concept inheritance could be seen as solution of this problem [11]. The notion of *concept* from [5] is extended in [6] to parametric models, where the parameters have both structure and behaviour. In this case we not only have a mechanism for the reutilisation of transformations, but mechanisms for the reutilisation and composition of models with behaviour.

**Multilevel transformation approaches.** An existing approach to multilevel model transformation rules which could be suitable for the definition of behaviour models is described in [2]. However, in the current implementation of this approach, only the instances which are at the lowest level in the metamodelling hierarchy will be transformed. A fine grained stack of languages requires a more flexible approach for the definition of their semantics.

**Reusable multilevel coupled model transformation.** In our approach, as depicted in Fig. 1b, we enforce that the metamodels of the behaviour languages are typed by a fixed top level metamodel $MM$. In this way the typing-by-composition is achieved without additional




(a) Generic metamodel (b) Petri Nets metamodel (c) CPN metamodel (d) TPN metamodel

Figure 2: Sample multilevel hierarchy in MultEcore

mappings to a concept metamodel. Moreover, one can modify the rules for the specific elements, and the modified rules will have the precedence. The realisation of our approach is planned to be implemented in the MultEcore metamodelling tool [8]. Fig. 2 outlines a multilevel hierarchy where an excerpt of the generic metamodel on the top is shown in 2a. As sample languages, metamodels for Petri Nets (PN), Coloured PN and Timed PN are shown in Fig 2b, 2c and 2d, respectively, where CPN and TPN are typed over PN. Similarities in PN, CPN and TPN can obviously be exploited by reusable rules, e.g. rules for token passing are quite similar in all three variants of PN. Currently we are working on a custom DSL and a transformation engine which implement our approach to reusable, multilevel coupled transformation rules.

## References

[1] C. Atkinson, R. Gerbig, and C. Tunjic. Towards Multi-level Aware Model Transformations. In Z. Hu and J. de Lara, editors, *In proceedings of ICMT*, pages 208–223. Springer, 2012.

[2] C. Atkinson, R. Gerbig, and C. V. Tunjic. Enhancing classic transformation languages to support multi-level modeling. *Software & Systems Modeling*, 14(2):645–666, 2015.

[3] C. Atkinson and T. Khne. Reducing accidental complexity in domain models. *Software & Systems Modeling*, 7(3):345–359, 2008.

[4] J. de Lara and E. Guerra. *Generic Meta-modelling with Concepts, Templates and Mixin Layers*, pages 16–30. Springer, 2010.

[5] J. de Lara and E. Guerra. Towards the flexible reuse of model transformations: A formal approach based on graph transformation. *JLAMP*, 83(56):427 – 458, 2014.

[6] F. Durán, A. Moreno-Delgado, F. Orejas, and S. Zschaler. Amalgamation of domain specific languages with behaviour. *JLAMP*, 2015. In Press (http://dx.doi.org/10.1016/j.jlamp.2015.09.005).

[7] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. Reuse in model-to-model transformation languages: are we there yet? *SoSyM*, 14(2):537–572, 2015.

[8] F. Macas, A. Rutle, and V. Stolz. MultEcore: Combining the best of fixed-level and multilevel metamodelling. In *MULTI*, CEUR Workshop Proceedings. CEUR-WS.org, 2016. To appear.

[9] R. Rodriguez-Echeverria and F. Macias. A statistical analysis approach to assist model transformation evolution. In *MODELS*, pages 226–235, Sept 2015.

[10] A. Rutle, W. MacCaull, H. Wang, and Y. Lamo. A metamodelling approach to behavioural modelling. In *Proceedings of BM-FA '12*, pages 5:1–5:10. ACM, 2012.

[11] J. Sánchez Cuadrado, E. Guerra, and J. de Lara. Generic Model Transformations: Write Once, Reuse Everywhere. In J. Cabot and E. Visser, editors, *ICMT*, pages 62–77. Springer, 2011.

# Pomset Languages of Higher-Dimensional Automata

Uli Fahrenberg

École Polytechnique, Palaiseau, France*

We are interested in languages of concurrent systems. Pomsets, generalizations of strings to partial orders, are well-suited to describe concurrent executions. Hence we shall be concerned with languages of pomsets.

To model the behavior of concurrent systems, we use higher-dimensional automata (HDA), for several reasons: they capture a number of other models for concurrency [6]; they are an intuitive generalization of standard automata; and they have a geometric realization as cubical complexes, which means that methods and techniques from algebraic topology can be applied [2].

We will thus be working with pomset languages of HDA. We proceed to define these notions. Let $\Sigma = \{a_1, \dots, a_m\}$ be a fixed finite set.

**Pomsets.** A *pomset* (over $\Sigma$) consists of a finite partially ordered set (a *poset*) $(P, \le)$ and a labeling $l : P \to \Sigma$. A *morphism* of pomsets $P = (P, \le_P, l_P)$ and $Q = (Q, \le_Q, l_Q)$ is a function $f : P \to Q$ which preserves the ordering and the labeling, *i.e.,* such that $x \le_P y$ implies $f(x) \le_Q f(y)$ and $l_Q \circ f = l_P$. This defines a category $\mathsf{Pomset}$.

A pomset is called *trivial* if its order is equality and *topological* if its order is total. Trivial pomsets are (finite) multisets; topological pomsets are (finite) strings. The *empty* pomset $\varepsilon$ is both trivial and topological.

Let $P = (P, \le, l)$ and $P' = (P, \le', l')$ be pomsets over the same set $P$. Following [5], we say that $P'$ is a *subsumption* of $P$, and write $P \succeq P'$, if there is a morphism $f : P \to P'$ which is the identity as a function $P \to P$. This defines a partial order $\succeq$ on pomsets. Note that topological and trivial pomsets are minimal, respectively maximal, in the order $\succeq$.

The *parallel product* [5] of pomsets $P = (P, \le_P, l_P)$, $Q = (Q, \le_Q, l_Q)$ is the pomset $P \otimes Q = (P \sqcup Q, \le, l)$, where for $v, v' \in P \cup Q$, $l(v) = l_P(v)$ if $v \in P$, $l(v) = l_Q(v)$ if $v \in Q$, and

$$v \le v' \quad \text{iff} \quad v \le_P v' \text{ or } v \le_Q v' \,.$$

Here $\sqcup$ denotes disjoint union. Note that every trivial pomset is a parallel product of one-element pomsets.

Together with *serial product*, see below, parallel product of pomsets has been used to generate a class of *series-parallel pomsets*. These are quite well-studied, for example in [1, 5]. We will *not* be concerned with series-parallel pomsets here.

We introduce a new type of (ternary) composition of pomsets. Let $P = (P, \le_P, l_P)$, $Q = (Q, \le_Q, l_Q)$ be pomsets and $R \subseteq P \cap Q$ such that for $v, v' \in R$, $v \le_P v'$ iff $v \le_Q v'$ and $l_P(v) = l_Q(v)$. The *gluing product of $P$ and $Q$ along $R$* is the pomset $P \overset{R}{\smile} Q = ((P \sqcup Q) \setminus R, \le, l)$, where for $v, v' \in P \cup Q$, $l(v) = l_P(v)$ if $v \in P$, $l(v) = l_Q(v)$ if $v \in Q$, and

$$v \le v' \quad \text{iff} \quad v \le_P v' \text{ or } v \le_Q v', \text{ or } v \in P \setminus R \text{ and } v' \in Q \setminus R \,.$$

This is a special type of pushout in the category $\mathsf{Pomset}$ (hence gluing product is associative). Gluing product $P \overset{\emptyset}{\smile} Q$, *i.e.,* with $R = \emptyset$, is the serial product of [5].

*Most of this work was carried out while the author was still employed by Inria Rennes, France.

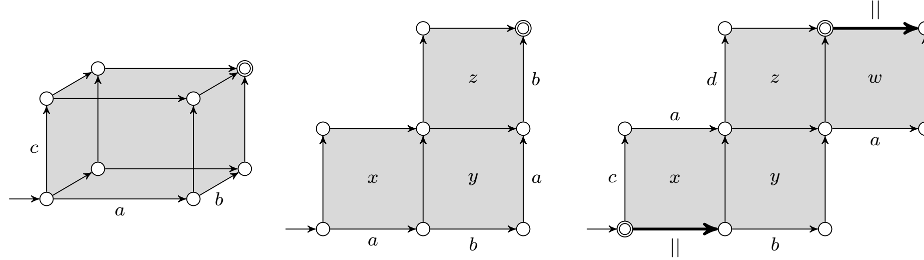



Figure 1: Some examples of HDA. Left, the 3-cube; center, three squares with autoconcurrency; right, four squares concatenated to a loop at the thick edges.

**HDA.** A *precubical set* is a graded set $X = \{X_n\}_{n\in\mathbb{N}}$ together with mappings $\delta_k^\nu : X_n \to X_{n-1}$, $k \in \{1,\dots,n\}$, $\nu \in \{0,1\}$, satisfying the *precubical identity*

$$\delta_k^\nu \delta_\ell^\mu = \delta_{\ell-1}^\mu \delta_k^\nu \qquad (k < \ell)\,.$$

We always assume the $X_n$ to be disjoint, *i.e.*, $X_n \cap X_m = \emptyset$ for $n \neq m$. The mappings $\delta_k^\nu$ are called *face maps*, and elements of $X_n$ are called *n-cubes*. A precubical set $X$ is *finite* if $\bigcup_{n\in\mathbb{N}} X_n$ is a finite set. *Morphisms* $f : X \to Y$ of precubical sets are graded functions $f = \{f_n : X_n \to Y_n\}_{n\in\mathbb{N}}$ which commute with the face maps: $\delta_k^\nu \circ f_n = f_{n-1} \circ \delta_k^\nu$ for all $n \in \mathbb{N}$, $k \in \{1,\dots,n\}$, $\nu \in \{0,1\}$. This defines a category $\mathsf{pCub}$.

We construct a special labeling precubical set $!\Sigma = \{!\Sigma_n\}$ out of $\Sigma = \{a_1,\dots,a_m\}$. Let $!\Sigma_n = \{(a_{i_1},\dots,a_{i_n}) \mid i_k \le i_{k+1} \text{ for all } k = 1,\dots,n-1\}$, with face maps defined by $\delta_k^\nu(a_{i_1},\dots,a_{i_n}) = (a_{i_1},\dots, a_{i_{k-1}}, a_{i_{k+1}},\dots,a_{i_n})$. A *higher-dimensional automaton* (HDA) is a structure $(X, I, F, \lambda)$, where $X$ is a finite precubical set, $I, F \subseteq X_0$ are the subsets of initial and accepting states, and $\lambda : X \to !\Sigma$ is a precubical morphism.

For $x, y \in X$ in a precubical set $X$, we write $x \lhd_0 y$ ($y \rhd_1 x$) if there is an index $k$ such that $x = \delta_k^0 y$ ($x = \delta_k^1 y$), and we let $\lhd_0^+$ and $\rhd_1^+$ be the transitive closures of these relations. A *run* in a HDA $(X, I, F, \lambda)$ is a sequence $(x_1,\dots,x_m)$ of elements of $X$ such that for all $i = 1,\dots,m-1$, $x_i \lhd_0^+ x_{i+1}$ or $x_i \rhd_1^+ x_{i+1}$. An *accepting run* is a run $(x_1,\dots,x_m)$ for which $x_1 \in I$ and $x_m \in F$.

We introduce pomset labels of runs. First, for $x \in X_n$ with $n \ge 1$ and $\lambda(x) = (a_{i_1},\dots,a_{i_n})$, $\bar\lambda(x)$ is the multiset $\{\!\{a_{i_1},\dots,a_{i_n}\}\!\}$; for $x \in X_0$, $\bar\lambda(x) = \emptyset$. Now let $\rho = (x_1,\dots,x_m)$ be a run in $X$, then an element $x_p$ is *maximal* (*minimal*) if $x_{p-1} \lhd_0^+ x_p \rhd_1^+ x_{p+1}$ ($x_{p-1} \rhd_1^+ x_p \lhd_0^+ x_{p+1}$). The maximal and minimal elements of $\rho$ form an alternating subsequence $(\hat x_1, \check x_1, \hat x_2,\dots,\check x_{p-1},\hat x_p)$, and then the *pomset label* of $\rho$ is

$$\bar\lambda(\rho) = \bar\lambda(\hat x_1) \overset{\bar\lambda(\check x_1)}{\smile} \bar\lambda(\hat x_2) \overset{\bar\lambda(\check x_2)}{\smile} \cdots \overset{\bar\lambda(\check x_{p-1})}{\smile} \bar\lambda(\hat x_p)\,.$$

A pomset $P$ is *accepted* by a HDA $X$ if there is an accepting run $\rho$ in $X$ with $\bar\lambda(\rho) = P$. The *language* of $X$ is $L(X) = \{\bar\lambda(\rho) \mid \rho \text{ accepting run in } X\}$.

**Examples.** We display some examples of HDA in Fig. 1. The 3-cube $X$ on the left is generated by one element $x \in X_3$; $X_2$ has six elements, $X_1$, twelve, and $X_0$, eight. Let $i = \delta_1^0\delta_1^0\delta_1^0 x$, $f = \delta_1^1\delta_1^1\delta_1^1 x$, and $I = \{i\}$, $F = \{f\}$. There are several accepting runs which contain $x$, for example $(i, x, f)$ or $(i, \delta_1^0 x, x, f)$. They all have pomset label $\{\!\{a, b, c\}\!\}$, the three-element multiset. There are also accepting runs which do not go through $x$, for example $(i, \delta_1^0 x, \delta_2^1 x, f)$ with pomset label

$$\begin{pmatrix} b \\ c \end{pmatrix} \overset{(c)}{\smile} \begin{pmatrix} a \\ c \end{pmatrix} = \begin{pmatrix} b \longrightarrow a \\ c \end{pmatrix}.$$

It can be shown that $L(X) = {\downarrow}\{\{\!\{a, b, c\}\!\}\}$, the *subsumption closure*.

The HDA in the center of Fig. 1 admits an accepting run $(i, x, \delta_1^1 x, y, \delta_2^1 y, z, f)$, which has pomset label

$$\begin{pmatrix} a \\ a \end{pmatrix} \overset{(a)}{\smile} \begin{pmatrix} a \\ b \end{pmatrix} \overset{(b)}{\smile} \begin{pmatrix} b \\ b \end{pmatrix} = \begin{pmatrix} a & \\ a \longrightarrow b \end{pmatrix} \overset{(b)}{\smile} \begin{pmatrix} b \\ b \end{pmatrix} = \begin{pmatrix} a \longrightarrow b \\ a \longrightarrow b \end{pmatrix} .$$

As series-parallel pomsets are *N-free* [5], this shows that pomset labels of HDA runs are not necessarily series-parallel.

The HDA to the right has infinitely many accepting runs. Some of their pomset labels are

$$\begin{pmatrix} a \longrightarrow b \\ c \longrightarrow d \end{pmatrix}, \begin{pmatrix} a \longrightarrow b \longrightarrow a \longrightarrow b \\ c \longrightarrow d \longrightarrow c \longrightarrow d \end{pmatrix}, \begin{pmatrix} a \longrightarrow b \longrightarrow a \longrightarrow b \longrightarrow a \longrightarrow b \\ c \longrightarrow d \longrightarrow c \longrightarrow d \longrightarrow c \longrightarrow d \end{pmatrix} .$$

**Properties.** We show that the pomsets accepted by HDA are precisely the *interval orders* $\mathcal{I} \subseteq \mathsf{Pomset}$ [4]. Recall that a pomset is an interval order iff is it *2+2-free*, *i.e.,* does not contain $( \rightrightarrows )$ as a sub-pomset.

We show that languages of HDA are subsumption-closed (or *weak* in the sense of [3]). We introduce *regular operations* on weak subsets of $\mathcal{I}$: union $L_1 \cup L_2$, gluing product $L_1 \overset{R}{\smile} L_2$, and gluing star $L \overset{R}{\smile}{}^*$, for $R \in \mathcal{T}$. These are defined by

$$L_1 \overset{R}{\smile} L_2 = {\downarrow}\{P \overset{R}{\smile} Q \mid P \in L_1, Q \in L_2, P \overset{R}{\smile} Q \text{ defined}\};$$
$$L \overset{R}{\smile}{}^* = \{\varepsilon\} \cup L \cup L \overset{R}{\smile} L \cup L \overset{R}{\smile} L \overset{R}{\smile} L \cup \cdots .$$

We show that the class $\mathsf{HReg}$ of languages of HDA is closed under these regular operations. Conversely, if $\mathsf{HRat} \subseteq 2^{\mathcal{I}}$ denotes the class generated by $\emptyset$ and $\{R\}$ for all $R \in \mathcal{T}$ and closed under the regular operations, then $\mathsf{HRat} = \mathsf{HReg}$.

We are working on an algebraic characterization of $\mathsf{HReg}$: using a new notion of *indexed semiring*, we conjecture that $\mathsf{HReg}$ is the free $\mathcal{T}$-indexed Kleene algebra and the free $\mathcal{T}$-indexed $^*$-continuous Kleene algebra. We are also working on notions of determinacy and completeness of HDA, in order to explore *complementation*.

**Acknowledgment.** I have started this work together with Zoltán Ésik when I visited him in Szeged in February 2016. Unfortunately, Zoltán did not live to see it completed.

## References

[1] Stephen L. Bloom and Zoltán Ésik. Free shuffle algebras in language varieties. *Theor. Comput. Sci.*, 163(1&2):55–98, 1996.

[2] Lisbeth Fajstrup, Martin Raussen, and Éric Goubault. Algebraic topology and concurrency. *Theor. Comput. Sci.*, 357(1-3):241–278, 2006.

[3] Jean Fanchon and Rémi Morin. Regular sets of pomsets with autoconcurrency. In Luboš Brim, Petr Jančar, Mojmír Křetínský, and Antonín Kučera, editors, *CONCUR*, volume 2421 of *Lect. Notes Comput. Sci.*, pages 402–417. Springer, 2002.

[4] Peter C Fishburn. Intransitive indifference with unequal indifference intervals. *J. Math. Psych.*, 7(1):144–149, 1970.

[5] Jay L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61:199–224, 1988.

[6] Rob J. van Glabbeek. On the expressiveness of higher dimensional automata. *Theor. Comput. Sci.*, 356(3):265–290, 2006. See also [7].

[7] Rob J. van Glabbeek. Erratum to "On the expressiveness of higher dimensional automata". *Theor. Comput. Sci.*, 368(1-2):168–194, 2006.

# Approximating Probabilities in Static Analysis

Maja H. Kirkeby *

Roskilde University, Denmark

## 1 Introduction

Program properties like resource properties (e.g. execution time) can often be expressed more precisely by functions over a measure of input (e.g. the measure gives the length of the list instead of the list itself). Static analysis typically derive such property-functions describing bounds. However, as Di Pierro et al. states *"A recent trend in system design is highlighting the need for formal analysis techniques that are able to provide quantitative estimates of a system property"* [3]. Such a static analysis may derive functions describing probability distributions of system properties. There are different approaches to this, typically divided by whether the programs analyzed have the Markov property [6] or not [7, 8, 5]; we will focus on the latter.

In the previous probabilistic analysis, such a bound has been defined to be a positive measure; a generalization of the probability measure where the total sum can be greater than 1. Due to the inherited additivity property, this generalization results in an unnecessary imprecision and lack of expressive power. In this abstract, we (i) present a novel and more well-suited way to interpret over-approximated (and under-approximated) probabilities in static analysis, and (ii) introduce and exemplify an approach arising from this interpretation keeping the same time complexity as the previous analyses. In the following will focus on output analyses, because these can be seen as a generalization of resource analysis.

## 2 Measures

Let $(\Omega, \mathcal{F})$ be a measurable space, and let $\mu : \mathcal{F} \to [0, \infty]$ be a set-function on this space satisfying the following properties for any $A, B \in \mathcal{F}$ with $A \cap B = \emptyset$: (1) $\mu(A) \geq 0$, (2) $\mu(A) + \mu(B) = \mu(A \cup B)$(*additivity*) where $A^{\mathrm{c}}$ denotes $\Omega \setminus A$, the complement of A. Then $\mu$ is a *positive measure*. A positive measure where $\mu(\Omega) = 1$ is a *probability measure*, typically denoted $P$. It holds that $P(A) = 1 - P(A^{\mathrm{c}})$, where $A^{\mathrm{c}}$ denotes $\Omega \setminus A$, the complement of A.Let $Bel, Pl : \mathcal{F} \to [0, 1]$ be set-functions on $(\Omega, \mathcal{F})$ satisfying the following properties for any $A, B \in \mathcal{F}$ with $A \cap B = \emptyset$: (1) $Bel(\emptyset) = Pl(\emptyset) = 0$, $Bel(\Omega) = Pl(\Omega) = 1$ (2) $Bel(A) + Pl(A^{\mathrm{c}}) = 1$ (3) $Bel(A) + Bel(B) \leq Bel(A \cup B)$ (Super-additivity) (4) $Pl(A) + Pl(B) \geq Pl(A \cup B)$ (Sub-additivity) Then $Bel$ and $Pl$ are *belief* and *plausibility* measures, respectively. For each Plausibility measure there exist a *dual* belief measure (and vice versa) for which it always holds that $Bel(A) \leq Pl(A)$ and $Bel(A) = 1 - Pl(A^{\mathrm{c}})$.

The *focal element mass* or simply *mass* is a single representation which expresses both the plausibility and belief measure. Let $(\Omega, \mathcal{F})$ be a measurable space then any belief function $Bel$ on $\mathcal{F}$ can be rewritten to its mass representation $m$ as follows: Let $\mathbf{A}$ be a set of focal elements s.t. $\mathcal{F} \supseteq \mathbf{A}$ and the following holds for all $A \in \mathbf{A}$:

$$m(A) = \textstyle\sum_{B \subseteq A} (-1)^{|A|-|B|} Bel(B) \quad \Leftrightarrow \quad Bel(A) = \textstyle\sum_{B \subseteq A} m(B)$$
$$\Leftrightarrow \quad Pl(A) = \textstyle\sum_{A \cap B \neq \emptyset} m(B).$$

*The research leading to these results has been supported by EU FP7 project 318337, ENTRA - Whole-Systems Energy Transparency and EU FP7 project 611004, Coordination and Support Action ICT-Energy.

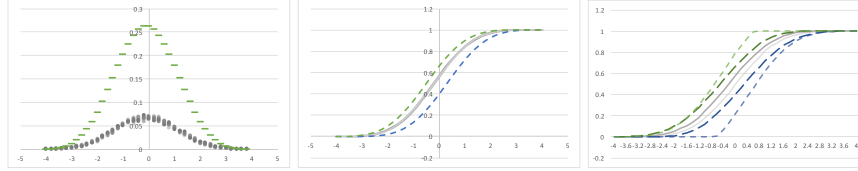


Figure 1: Lines: grey are simulations, green and blue are upper and lower bounds, resp. (Left): Monniaux's results, (Middle): Reinterpreting Monniaux's results as focal elements, and (Right): Our improved results (long dash) and Monniaux's (short dash) of the altered test-program.

## 3 Induced Evidence

In a probabilistic analysis of a program, we are given the probability of the input (the random generators are seen as input) and derive a set of possible outputs along with their probabilities. If we see the analysis as the relation from the input to the related possible output, we can use Dempster's theory of induced lower and upper probabilities [2] and derive lower and upper probabilities for the program output. This relation is a multivalued mapping, defined as follows.

**Definition 1** (Multivalued mappings and induced probabilities)**.** *Given pair of measurable sets* $(X, \mathcal{X}), (S, \mathcal{S})$*, a* multivalued *mapping* $\Gamma : X \to \mathcal{S}$ *associates each element of* $X$ *to a subset of* $S$*. The domain of* $\Gamma$*,* $\text{dom}(\Gamma)$*, is defined as* $\text{dom}(\Gamma) = \{x \mid x \in X \wedge \Gamma(x) \neq \emptyset\}$*. The (mapping) image of a subset* $A \in \mathcal{X}$ *is* $\Gamma(A) = \bigcup_{a \in A} \Gamma(a)$*.*

*Furthermore, let* $\mu$ *be a probability distribution over* $\mathcal{X}$*, then* $m$ *is the* mass representation *over* $\mathcal{S}$*:* $m(A) = \mu(\{x \in X \mid x \in \text{dom}(\Gamma), \Gamma(x) = A\})$*.*
*When* $X$ *and* $S$ *are finite,* $m$ *uniquely defines a pair of dual belief and Plausibility measure[2].*

Recall, that belief and Plausibility over $S$ can be defined via the mass.

**Lemma 1.** *Given that the transfer function* $\Gamma : \mathcal{X} \to \mathcal{S}$ *overapproximates the program execution* $\texttt{prg}\colon X \to S$ *for all program inputs, i.e.* $\forall x \in X : \texttt{prg}(x) \in \Gamma(x)$ *then the following holds. The Plausibility measure is an overapproximation of the programs output distribution, and the belief measure is an underapproximation of the programs output measure.*

*Proof.* Let the input probability be $P_{\text{IN}}$, and the output probability of a program be $P_{\texttt{prg}}$.

$$P_{\texttt{prg}}(s) = \sum_{x \in X : \texttt{prg}(x) = s} P_{\text{IN}}(x) \leq \sum_{x \in X : x \in \text{dom}(\Gamma) \wedge \Gamma(x) \cap \{s\} \neq \emptyset} P_{\text{IN}}(x) = \sum_{B \cap \{s\} \neq \emptyset} m(B) = Pl_S(\{s\})$$

$$P_{\texttt{prg}}(s) = \sum_{x \in X : \texttt{prg}(x) = s} P_{\text{IN}}(x) \geq \sum_{x \in X : x \in \text{dom}(\Gamma) \wedge \Gamma(x) \subseteq \{s\}} P_{\text{IN}}(x) = \sum_{B \subseteq \{s\}} m(B) = Bel_S(\{s\}) \qquad \square$$

Equivalently plausibility and belief can be induced [4].

## 4 Results

In 2000, Monniaux presented [7] statically derived upper bounds for the probability distribution of the output of the following program (original program) which adds four independent real variables (each uniformly distributed in $[0, 1]$). His work is state-of-the-art, and we use this program and an altered version with a test to demonstrate the power of the novel understanding of over-approximations and our improvements of the existing analysis.

```
double x=0.0;
int i;
for (i=0; i<4; i++)
    x += drand48()*2.0-1.0;
```
(original program)

```
double x=0.0;  int i;
for (i=0; i<4; i++){
    if ((double) i > drand48()-0.5){
    x += drand48()*2.0-1.0;}}
```
(altered program)

Here, `drand48()` returns reals uniformly distributed in $[0, 1]$. In order to create a measurable set from the reals, Monniaux abstracts the reals into intervals and calculates a probability for the interval $([n/N, (n+1)/N], 1/N)_{0\leq n<N}$ where $N$ is a parameter. He defines the output measure to be a positive measure. The concrete domain $\langle \wp(\mathbb{Z}), \subseteq\rangle$ is connected with the abstract domain, the interval domain $\langle I(\mathbb{Z}), \sqsubseteq_I\rangle$ where $I(\mathbb{Z}) = \bot \cup \{[a, b] \mid a \leq b\}$, with the following abstraction and concretization: $\alpha(A) = [\min(A), \max(A)]$ and $\alpha(\emptyset) = \bot$, where $\min(\mathbb{Z}) = -\infty$ and $\max(\mathbb{Z}) = \infty$. $\gamma([a, b]) = \{c \mid a \leq c \leq b \wedge c \in \mathbb{Z}\}$ and $\gamma(\bot) = \emptyset$. Let `a+b` = `a`+`b` and `[a,b]+[c,d]` = `[a`+`c, b`+`d]` be the concrete and abstract definition of addition, respectively. We use these domains when analysing both programs.

With a light implementation we have reproduced Monniauxs results ($N = 10$) [7] as depicted in Figure 1 (Left). Here, the grey dots are program simulations, and the lines are Monniaux's statically derived upper bounds. Monniaux points out and correctly describes the results: "An intuitive vision of this somewhat paradoxical behavior is that our abstract domain represents masses quite exactly, but loses precision on their exact location" [7].

Probabilistic results are often unnecessarily concretized to a probability distribution to relate them with the actual behaviour. This can also be achieved using cumulative distributions [1]. It is novel to exploit the duality of belief and plausibility measures to deduce lower bounds even when the analysis deducts upper bounds. Their duality allows us to reason about the complement set, whereas positive measures which are unbounded do not have this quality. We let Monniaux's overlapping output intervals (not the depicted intervals) be the focal elements, and their probability be the mass; the lower and upper bound are depicted in Figure 1 (Middle).

The analyses [7, 1] treat tests in an imprecise way; for each result, they sum the probabilities of reaching that result for each branch. Instead, we calculate a pair of plausibility and beliefs for each result considering both the branches, i.e., calculating $\Gamma$ and then $m$. The promising new results for the altered program is shown by the long dashes in Figure 1 (Right).

# References


[1] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault, and S. Putot. Static analysis of programs with imprecise probabilistic inputs. In *In Verified Software: Theories, Tools, Experiments*, pages 22–47. Springer Berlin Heidelberg., 2014.

[2] A. P. Dempster. Upper and lower probabilities induced by a multivalued mapping. *Ann. Math. Statist.*, 38(2):325–339, 04 1967.

[3] A. Di Pierro and H. Wiklicky. Semantics of Probabilistic Programs: A Weak Limit Approach. pages 241–256. 2013.

[4] D. Dubois and H. Prade. Evidence measures based on fuzzy information. *Automatica*, 21(5):547–562, 1985.

[5] M. H. Kirkeby and M. Rosendahl. Probabilistic Resource Analysis by Program Transformation. 2015. Accepted FOPARA 2015 http://adsabs.harvard.edu/abs/2016arXiv160801106K.

[6] M. Kwiatkowska, G. Norman, and D. Parker. *PRISM 4.0: Verification of Probabilistic Real-Time Systems*, pages 585–591. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[7] D. Monniaux. Abstract Interpretation of Probabilistic Semantics. In J. Palsberg, editor, *SAS*, volume 1824 of *LNCS*, pages 322–339. Springer, 2000.

[8] M. Rosendahl and M. H. Kirkeby. Probabilistic Output Analysis by Program Manipulation. *Electronic Proceedings in Theoretical Computer Science*, 194(318337):110–124, 2015.

# An Operational Semantics for Multicasting Systems with Monotonic Values *

Seyed H. Haeri (Hossein) and Peter Van Roy

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

**Abstract**

We present an operational semantics as a simplified model for edge computing: Nodes can leave and join any time; each node performs computations independently, but, can also wait for values to arrive from peers; nodes can multicast to one another; and, most importantly, values can only grow monotonically. We prove an eventual consistency result for our operational semantics.

## 1 Introduction

In edge computing, nodes are only loosely coupled. They are very dynamic on when they join or leave. A node, also, typically has limited computational power and resource. General computations, however, need to take place on edge devices. Yet, there are still no many formal models available which can truly describe edge computing. Reasoning about edge computing is, hence, not quite possible these days.

To that end, we present an operational semantics in this paper that is parametrised by the semantics of computations done at individual nodes. Simplicity is a key feature of our operational semantics that facilitates reasoning. Nodes, in our model, can leave and join at any time. Each node can contribute their own part to a general computation or wait for peers to come back with the results of their own part of computation. At each step, each node receives a new instruction. When to take each instruction is a property of each node that is only deterministic to the node itself. Besides, each node can choose to multicast values to peers. Values are only allowed to increase monotonically so that need rises for synchronisation.

Under certain common fairness properties, we show how every set of nodes is eventually consistent: Theorem 1. Due to space restrictions, we drop technicality – especially, on the proof of Theorem 1.

The research in this paper is amongst the last steps taken over the SyncFree project. The need for this research was clearly indicated over the last year of SyncFree. The research in the paper is to be continued over a follow-up H2020 project called LightKone that has only very recently been accepted. This paper and its accompanying talk are to welcome discussion and feedback on our early developments.

## 2 Settings

$\nu$ ranges over nodes. We take the syntax of expressions (ranged over by $e, e', \dots, e_1, e_2, \dots$) and values (ranged over by $v, v', \dots, v_1, v_2, \dots$) for granted. We single out a class of values $\circ$ for expressions with *lack*ing variables. (See below for more.) The subset $\circ/x$ of $\circ$ represents those elements of the latter set specifically lacking $x$, over which $\circ_x$ ranges.[1] Instructions are

*Funded by the Free project of the European Seventh Framework Programme under Grant Agreement 609551.

[1] Although similar to Nominal Set Theory [6], this is different because $\circ_x$ has a structure too.

ranged over by $\iota$ taking the forms $\mathsf{run}\ e \mid \mathsf{bind}\ x\ \mathsf{to}\ v \mid \mathsf{mcast}\ x\ \mathsf{to}\ \{\overline{\nu}\}$. Let $I$ range over sets of instructions. The instruction that corresponds to $\nu$ in $I$ is denoted by $I_\nu$. $N$ ranges over sets of nodes (denoted by $\{\overline{\nu}\}$) that we call hordes.

Each node $\nu$ has a *configuration* $c(\nu)$ that consists of the quadruple $\langle T, W, B, S\rangle$, for *ToTake*, *WaitFor*, *Broadcast* repository, and *Store*, respectively: the instructions for a node to do; the expressions waiting for the arrival of new information; the bindings broadcast (by peers in the same horde); and, a mapping of variables to values. We write $c_N(\nu)$ for $\nu$'s configuration when $\nu \in N$. When a node $\nu$ evolves from a configuration $c_1$ to a configuration $c_2$, we write $\nu \vdash c_1 \to c_2$. When the node of discourse is clear, we drop the "$\nu \vdash$" portion and just write $c_1 \to c_2$. We break the monolithic description of a node evolution $\langle T, W, B, S\rangle \to \langle T', W', B', S'\rangle$ into the collection of four smaller elements $\{T \to T', W \to W', B \to B', S \to S'\}$, each element of which we will display in a separate line. Inspired by I-MSOS [5], that gives us the freedom of leaving out the elements of a configuration that remain untouched over a node evolution.

We assume the availability of an expression evaluation "$\Downarrow$" with the schema $S_e : e \Downarrow S_v : v$, reading: 'In a store $S_e$, the expression $e$ reduces to the value $v$ with the store $S_v$ that is possibly updated over the evaluation.' The value produced by $\Downarrow$ can also be $\circ_x$, in which case, $\Downarrow$ signals it that a new binding for $x$ is required for it to continue the evaluation. With the complexity allowed for $\circ_x$, we reserve $r(\circ_x)$ for $\circ_x$ modified so it is known to $\Downarrow$ how to *resume* the original computation from where left. A necessary condition for $\Downarrow$ is that it can only modify bindings **monotonically** over expression evaluation. Note, then, that it is permissible for $S_e : e \Downarrow S_v : v$ that $dom(S_v) \supset dom(S_e)$. Another necessary condition is that the exact same "$\Downarrow$" should be provided to each and every node. Finally, it should be that "$\Downarrow$" is deterministic in the following sense [3, 4]: $\forall S_e, e.\ [(S_e : e \Downarrow S_v : v) \wedge (S_e : e \Downarrow S'_v : v')] \Rightarrow [S_v = S'_v \wedge v = v']$.

We use the notation $T + \iota$ for $T$ augmented with the instruction $\iota$. When over an evolution $T + \iota \to T$, it is the task $\iota$ that has had the top priority and that is done over the evolution. Note that multiple instances of a single instruction might coexist in a single $T$. When $c(\nu) = \langle T, W, B, S\rangle$, we write $c(\nu) + \iota$ for $\langle T + \iota, W, B, S\rangle$

# 3 Nodes and Hordes

The node operational semantics is defined by the rules of Fig. 1, where the schema is $\nu \vdash c_1 \to c_2$. We call the the transition from $c_1$ to $c_2$ an action [1]. Let $\alpha, \alpha', \ldots, \alpha_1, \alpha_2, \ldots$ range over actions. When $\nu$ takes the action $\alpha$ to get from $c_1$ to $c_2$, we write $\nu \vdash c_1 \xrightarrow{\alpha} c_2$. When an instance of a rule (R) is used for the action $\alpha$ to take place, we write $\alpha \in$ (R). When $\alpha \in$ (E-DONE) or $\alpha \in$ (E-SUSP), we write $\alpha \in$ (E-*). We use a similar convention for (B-*) and (N-*).

Similar to Bloom$^L$ [2], we model the evolution of a system using the concept of timesteps: Let $N$ and $N'$ be hordes. Write $N \overset{I}{\Rightarrow} N'$ – read $N$ evolves to $N'$ – when: (i) $|N| = |I|$, and (ii) $\forall \nu \in N \cap N' \exists \alpha_\nu.\ \nu \vdash c_N(\nu) + I_\nu \xrightarrow{\alpha_\nu} c_{N'}(\nu)$. We drop $I$ when not required and write $N \Rightarrow N'$. For $\#\overline{I} = n > 0$, we write $N \overset{\overline{I}}{\Rightarrow}{}^* N'$ for $N = N_0 \overset{I_1}{\Rightarrow} \ldots \overset{I_n}{\Rightarrow} N_n = N'$. When $\overline{I}$ is not important in a context, we drop it and write $N \Rightarrow^* N'$. In such a case, we write $\#(N \Rightarrow^* N')$ for $\#\overline{I}$.

Let $\nu \in N$ and $\nu' \in N'$. Suppose also that $c_N(\nu) = \langle _, _, _, S\rangle$ and $c_{N'}(\nu') = \langle _, _, _, S'\rangle$. Define $\Delta_{N,N'}(\nu, \nu') = \{x \in dom(\nu) \cap dom(\nu') \mid S(x) \neq S'(x)\}$. Let $\Delta(N, N') = \sum_{\nu \in N, \nu' \in N'} \Delta_{N,N'}(\nu, \nu')$. When $N = N'$, we simply write $\Delta_N$. Moreover, for the suitable $k$, we write $\Delta_k$ for $\Delta_{N_k}$.

We call an action $\alpha$ generative when: $\alpha \in$ (N-*); or, $\alpha \in$ (E-*) and $\alpha$ modifies store. Call $\alpha$ nongenerative otherwise. When every action in a horde evolution $N \Rightarrow N'$ is nongenerative and $N' \subseteq N$, we call the evolution itself nongenerative. Likewise, call $N \Rightarrow^* N'$ nongenerative

$$\frac{S_e : e \Downarrow S_v : v}{\begin{cases} T + \mathsf{run}\ e \to T \\ S_e \qquad \to S_v \end{cases}}\ (\text{E-Done}) \qquad \frac{S_e : e \Downarrow S_\circ : \bigcirc_x}{\begin{cases} T + \mathsf{run}\ e \to T \\ W \qquad \to W + \bigcirc_x \\ S_e \qquad \to S_\circ \end{cases}}\ (\text{E-Susp})$$

$$\frac{x \notin dom(S) \quad \bigcirc_x \notin W}{\begin{cases} (B, x \mapsto v) \to B \\ S \qquad \to (S, x \mapsto v) \end{cases}}\ (\text{B-NW}) \qquad \frac{S(x) = v \quad W = W' + \bigcirc_x}{\begin{cases} (B, x \mapsto v') \to B \\ S \qquad \to S[x \mapsto v \sqcup v'] \\ W \qquad \to W' \\ T \qquad \to T + \mathsf{run}\ r(\bigcirc_x) \end{cases}}\ (\text{B-Wait})$$

$$\frac{x \notin dom(S) \quad \bigcirc_x \notin W}{\begin{cases} T + \mathsf{bind}\ x\ \mathsf{to}\ v \to T \\ S \qquad \to (S, x \mapsto v) \end{cases}}\ (\text{N-NW}) \qquad \frac{x \notin dom(S) \quad W = W' + \bigcirc_x}{\begin{cases} T + \mathsf{bind}\ x\ \mathsf{to}\ v \to T + \mathsf{run}\ r(\bigcirc_x) \\ S \qquad \to (S, x \mapsto v) \\ W \qquad \to W' \end{cases}}\ (\text{N-Wait})$$

$$\frac{\begin{matrix} \nu \in N \quad \{\overline{\nu}\} \subseteq N \quad \nu' \in \{\overline{\nu}\} \\ v = S_\nu(x) \qquad v' = B_{\nu'}(x) \end{matrix}}{\begin{cases} \nu \ \vdash T_\nu + \mathsf{mcast}\ x\ \mathsf{to}\ \{\overline{\nu}\} \to T_\nu \\ \nu' \vdash B_{\nu'} \qquad \to B_{\nu'}[x \mapsto v \sqcup v'] \end{cases}}\ (\text{MCast})$$

Figure 1: The Node Operational Semantics

when, for every $1 \le i \le n$, the evolution $N_{i-1} \Rightarrow N_i$ is nongenerative, where $n = \#(N \Rightarrow^* N')$.

For a node $\nu$, write $m_\nu(x) = m$ when $x$ is guaranteed to be multicast at least once every $m$ actions at $\nu$. When no such guarantee is at hand for $\nu$, we write $m_\nu(x) = \aleph_0$. Write $r_\nu(x) = r$ when $x$ is guaranteed to be committed at least once every $r$ actions at $\nu$. We use $r_\nu(x) = \aleph_0$ similar to $m_\nu(x)$. Call a node $\nu$ fair on $x$ when $m_\nu(x) < \aleph_0$ and $r_\nu(x) < \aleph_0$. Write $p_\nu(\nu') = p$ when $\nu$ is guaranteed to multicast to a peer $\nu'$ at least once every $p$ actions. Call a horde $N$ all-fair if every node $\nu \in N$ is fair on every variable and $p_\nu(\nu') < \aleph_0$ for every $\nu' \in N \setminus \{\nu\}$.

**Theorem 1** (Eventual Consistency)**.** *Let $N$ be an all-fair horde. Suppose that $N \Rightarrow^* N'$ be nongenerative and $n = \#(N \Rightarrow^* N')$. Then, $\lim_{n\to\infty} \Delta_n = \varnothing$.*

## References

[1] A. Bouajjani, C. Enea, and J. Hamza. Verifying Eventual Consistency of Optimistic Replication Systems. In S. Jagannathan and P. Sewell, editors, *POPL*, pages 285–296. ACM, Jan 2014.

[2] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and Lattices for Distributed Programming. In M. J. Carey and S. Hand, editors, *SCC*, page 1, Oct 2012.

[3] S. H. Haeri. Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness. In Z. Majkic, S.-Y. Hsieh, J. Ma, I. M. M. El Emary, and K. S. Husain, editors, *TMFCS*, pages 143–150. ISRST, Jul 2010.

[4] S. H. Haeri and S. Schupp. Distributed Lazy Evaluation: A Big-Step Mechanised Semantics. In $1^{st}$ *4PAD*, pages 751–755. IEEE, Feb 2014.

[5] P. D. Mosses and M. J. New. Implicit Propagation in Structural Operational Semantics. *ENTCS*, 229(4):49–66, 2009.

[6] A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science.* Cambridge University Press, New York, NY, USA, 2013.

# Reductions for transition systems at work: the bisimulation hierarchy of state-to-function transition systems*

Marino Miculan and Marco Peressotti

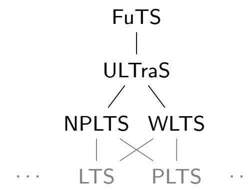
Department of Mathematics, Computer Science and Physics, University of Udine, Italy
marino.miculan@uniud.it marco.peressotti@uniud.it

## 1 Motivation

*Weighted labelled transition systems* (WLTSs) [8] is a meta-model for systems with quantitative aspects; transitions $P \xrightarrow{a,w} Q$ are labelled with actions $a$ and weights $w$, taken from some weighting structure. Many computational aspects can be captured just by changing the underlying weight structure: weights can model probabilities, resource costs, stochastic rates, *etc*. Hence, WLTSs are a generalisation of labelled transition systems (LTSs), probabilistic systems (PLTSs) [15], stochastic systems [7], among others. Definitions and results developed in this setting instantiate to existing models, thus recovering known results and discovering new ones. In particular, the notion of *weighted bisimulation* [8] in WLTSs coincides with strong bisimulation for all the aforementioned models.

In the wake of these encouraging results, other meta-models have been proposed aiming to cover an even wider range of computational models and concepts. *Uniform labelled transition systems* (ULTraSs) [1] are systems whose transitions have the form $P \xrightarrow{a} \phi$, where $\phi$ is a *weight function* assigning weights to states; hence, ULTraSs can be seen both as a non-deterministic extension of WLTSs and as a generalisation of Segala's probabilistic systems [14] (NPLTSs). In [11,12] a (coalgebraically derived) notion of bisimulation for ULTraSs is presented and shown to precisely capture bisimulations for weighted and Segala systems. *Function-to-state transition systems* (FuTSs) were introduced in [4] as a generalisation of the above, of IMC [6], and other models. Later, [9] defines a (coalgebraically derived) notion of bisimulation for FuTSs which instantiates to known bisimulations for the aforementioned models.

One then wonders about the *expressiveness* of these meta-models. Previous work [1,8,9,11,12] have shown that, with respect to strong bisimulation, each of these meta-models is a conservative extension of the previous ones, thus forming the hierarchy shown aside. Intuitively, a meta-model $\mathsf{M}$ is *subsumed* by $\mathsf{M}'$ if any system $S$ which is an instance of $\mathsf{M}$ is an instance also of $\mathsf{M}'$, provided the two agree on strong bisimulations (i.e., two states from the first instance are bisimilar if and only if their counterparts from the second instance are bisimilar).




Still an important question remains open: is any of these meta-models *strictly more expressive* than others? In this work we address this question, proving that this is not the case: the black part of the hierarchy collapses!

In order to formally capture the notion of "expressiveness order between system classes with respect to (strong) bisimulation", we introduce the notion of *reduction* between classes of systems. Although the driving motivation is the study of the FuTS hierarchy under the lens of bisimulation, the notion of reduction is more general and, as defined in this work, can be used to study any class of state-based transition systems. In fact, all the constructions and results are developed abstracting from the "shape" of computation under scrutiny.

*This work is based on [13].

## 2 Reductions for discrete transition systems

As usual, systems are characterised as coalgebra for endofunctors over $\mathsf{Sets}$. In particular, FuTSs are coalgebras for functors generated by the grammar

$$T ::= (S-)^A \mid T \times (S-)^A \qquad S ::= \mathcal{F}_M \mid \mathcal{F}_M \circ S \tag{1}$$

where $A$ is a (non-empty) sets of labels, $M$ is a (non-trivial) abelian monoid, and $\mathcal{F}_M$ is given on any set $X$ and function $f\colon X \to Y$ as:

$$\mathcal{F}_M X = \{\phi\colon X \to M \mid \phi(x) \neq 0 \text{ on finitely many } x\} \qquad \mathcal{F}_M(f)(\phi) = \lambda y. \textstyle\sum_{x \in f^{-1}(y)} \phi(x).$$

Clearly, LTS, WLTSs, and ULTraSs are particular cases of FuTSs, since they are captured by endofunctors $(\mathcal{P}_f-)^A \cong (\mathcal{F}_{\mathbb{B}}-)^A$, $(\mathcal{F}_M-)^A$, and $(\mathcal{F}_{\mathbb{B}}\mathcal{F}_M-)^A$, respectively.

For $\alpha\colon X \to TX$ let $\mathrm{car}(\alpha)$ be $X$ and $\mathrm{bis}(\alpha)$ be the set of all (kernel) bisimulations for $\alpha$.

Intuitively, a reduction for a system is a mapping from its carrier to that of the target system such that it preserves and reflects bisimulations.

**Definition 1.** For systems $\alpha$ and $\beta$, of possibly different types, a *(system) reduction* $\sigma\colon \alpha \to \beta$ is given by a function $\sigma^c\colon \mathrm{car}(\alpha) \to \mathrm{car}(\beta)$ and a correspondence $\sigma^b \subseteq \mathrm{bis}(\alpha) \times \mathrm{bis}(\beta)$ such that $\sigma^c$ carries a relation homomorphism for any pair of bisimulations in $\sigma^b$, *i.e.*:

$$R\ \sigma^b\ R' \implies (x\ R\ x' \iff \sigma^c(x)\ R'\ \sigma^c(x')). \tag{2}$$

**Definition 2.** For categories of system $\mathsf{C}$ and $\mathsf{D}$, a *reduction* $\sigma\colon \mathsf{C} \to \mathsf{D}$, is a mapping that

1. assigns to any system $\alpha$ in $\mathsf{C}$ a system $\sigma(\alpha)$ in $\mathsf{D}$ and a system reduction $\sigma_\alpha\colon \alpha \to \sigma(\alpha)$;
2. assigns to any homomorphism $f\colon \alpha \to \beta$ in $\mathsf{C}$ an homomorphism $\sigma(f)\colon \sigma(\alpha) \to \sigma(\beta)$ in $\mathsf{D}$ such that: $\sigma^c_\beta \circ f = \sigma(f) \circ \sigma^c_\alpha$, $\sigma(id_\alpha) = id_{\sigma(\alpha)}$, and $\sigma(g \circ f) = \sigma(g) \circ \sigma(f)$.

A category of systems $\mathsf{C}$ is said to *reduce* to $\mathsf{D}$, if there is a reduction from the $\mathsf{C}$ to $\mathsf{D}$.

Some reductions can be systematically derived from the structure of the underlying endofunctors. For instance, a reduction from $(\prod_{i\in I} T_i)$-systems to $(\prod_{i \in I} S_i)$-systems can be given component-wise by means of a reduction for each pair $(T_i, S_i)$ and such that they all agree on the functions underlying system homomorphisms. Another example are reductions from $(T^{n+1})$-systems to $T$-systems. Intuitively, these can be seen as combining $(n+1)$-steps of the same kind into one by "hiding" intermediate steps inside the behaviour and thus a $T$-system can be obtained by singling out all intermediate steps and states. This suggests the following reduction: For $\alpha\colon X \to T^{n+1}X$ define $\underline{\alpha}\colon \underline{X} \to T(\underline{X})$ such that

$$\underline{X} = \coprod_{i=0}^{n} T^i X \qquad \underline{\alpha}(y) = \begin{cases} \alpha(y) & \text{if } y \in X \\ y & \text{if } y \in T^{i+1}X \end{cases}$$

Finally, any injective natural transformation $\mu\colon T \to S$ yields a reduction.

A combination of the above three techniques allows us to reduce FuTSs to WLTSs since:

- any $T$ generated by (1) is in fact a product of $S = \mathcal{F}_{M_1} \circ \cdots \circ \mathcal{F}_{M_n}$;
- any $S = \mathcal{F}_{M_1} \ldots \mathcal{F}_{M_n}$ is a subfunctor of $(\mathcal{F}_M)^n$ where $M$ is the product monoid $\prod_{i=1}^n M_i$;
- any $\prod_{i=1}^n \mathcal{F}_{M_i}$ is isomorphic to $\mathcal{F}_M$ where $M$ is the product monoid $\prod_{i=1}^n M_i$.

**Theorem 1.** *The FuTSs category reduces to that of simple FuTSs* i.e.*, WLTSs.*

*Proof.* See [13, Sec. 6]. □

## 3 Concluding remarks

The notion of *reduction* for categories of transition systems allows for the sound and systematic investigation of expressiveness of different meta-models. As an application of this framework we have shown that FuTSs reduce to WLTSs, as far as strong bisimulation is preserved.

Besides the classification interest, this result offers a solid bridge for porting existing and new results from WLTSs to FuTSs. For instance, SOS specifications formats presented in [8,12] can cope now with FuTSs, and any abstract GSOS for these systems admits a specification in the format presented in [12]. Likewise, an HML style logic for bisimulation on WLTSs would readily yield a logic capturing bisimulation on FuTSs.

It remains an open question whether the hierarchy can be further collapsed, especially when other notions of reduction are considered. In fact, requiring a correspondence between bisimulations for the source and target systems may be too restrictive in some applications such as bisimilarity-based verification techniques. This suggests to investigate laxer notions of reductions that *e.g.*, relax either direction of (2). Another direction is to consider other behavioural equivalences, like trace equivalence or weak bisimulation. We remark that, in order to deal with these and similar equivalences, endofunctors need a monadic (sub)structure [2, 3, 5]; although WLTSs are covered in [2, 10], a similar account of FuTSs is still an open problem.

## References

[1] M. Bernardo, R. De Nicola, and M. Loreti. A uniform framework for modeling nondeterministic, probabilistic, stochastic, or mixed processes and their behavioral equivalences. *Inf. Comput.*, 225:29–82, 2013.

[2] T. Brengos, M. Miculan, and M. Peressotti. Behavioural equivalences for coalgebras with unobservable moves. *JLAMP*, 84(6):826–852, 2015.

[3] T. Brengos and M. Peressotti. A Uniform Framework for Timed Automata. In *Proc. CONCUR*, volume 59 of *LIPIcs*, pages 26:1–26:15, 2016.

[4] R. De Nicola, D. Latella, M. Loreti, and M. Massink. A uniform definition of stochastic process calculi. *ACM Computing Surveys*, 46(1):5, 2013.

[5] I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace semantics via coinduction. *LMCS*, 3(4), 2007.

[6] H. Hermanns. *Interactive Markov Chains: The Quest for Quantified Quality*, volume 2428 of *LNCS*. Springer, 2002.

[7] J. Hillston. *A compositional approach to performance modelling*. Cambridge, 1996.

[8] B. Klin and V. Sassone. Structural operational semantics for stochastic and weighted transition systems. *Inf. Comput.*, 227:58–83, 2013.

[9] D. Latella, M. Massink, and E. P. de Vink. Bisimulation of labelled state-to-function transition systems coalgebraically. *LMCS*, 11(4), 2015.

[10] M. Miculan and M. Peressotti. Weak bisimulations for labelled transition systems weighted over semirings. *CoRR*, abs/1310.4106, 2013.

[11] M. Miculan and M. Peressotti. GSOS for non-deterministic processes with quantitative aspects. In N. Bertrand and L. Bortolussi, editors, *Proc. QAPL*, volume 154 of *EPTCS*, pages 17–33, 2014.

[12] M. Miculan and M. Peressotti. Structural operational semantics for non-deterministic processes with quantitative aspects. *TCS*, 2016.

[13] M. Miculan and M. Peressotti. On the bisimulation hierarchy of state-to-function transition systems. Accepted at *ICTCS*, 2016.

[14] R. Segala and N. A. Lynch. Probabilistic simulations for probabilistic processes. *Nord. J. Comput.*, 2(2):250–273, 1995.

[15] R. J. van Glabbeek, S. A. Smolka, and B. Steffen. Reactive, generative and stratified models of probabilistic processes. *Inf. Comput.*, 121:130–141, 1990.

# Finite tree automata determinisation and its application in program analysis and verification (extended abstract)*

John P. Gallagher[1,2], Mai Ajspur[3], and Bishoksan Kafle[1,4]

[1] Roskilde University, Denmark
[2] IMDEA Software Institute, Madrid
[3] IT University of Copenhagen
[4] The University of Melbourne, Australia
Email: {jpg,kafle}@ruc.dk, ajspur@itu.dk

*Introduction.* Determinisation is an important concept in the theory of finite tree automata (FTA). There are applications of tree automata in program analysis and verification, since descriptions of sets of terms (or trees) can model computational entities such as data structures, computation trees and proof trees. FTA have desirable properties such as closure under Boolean operations, and decidability of membership and emptiness. Tree automata have been applied in static analysis e.g. [2,12,15,4,10,11] and in automatic program verification [14]. Some important operations and properties of tree automata are stated in terms of bottom-up deterministic tree automaton (DFTA) [3]. However, the transformation to DFTA can result in an explosion of states and transitions; some previous attempts to use DFTA directly in static analysis reported problems with scalability [15,13]. The complexity of the textbook procedure for determinisation prevents it from being practical, even for quite small tree automata. In this work we show firstly that the standard algorithm for determinisation can be greatly improved; then we summarise some of the applications of tree automata that are enabled by a practical algorithm for determinisation.

*Finite Tree Automata.* An FTA is a quadruple $\langle Q, Q_f, \Sigma, \Delta\rangle$, where $Q$ is a finite set called *states*, $Q_f \subseteq Q$ is called the set of accepting (or final) states, $\Sigma$ is a set of function symbols and $\Delta$ is a set of *transitions*. Each transition in $\Delta$ is of the form $f(q_1, \ldots, q_n) \to q$, where the arity of $f$ is $n$ and $q, q_1, \ldots, q_n \in Q$. A tree automaton $R$ defines a set of terms denoted $L(R)$, as the set of all terms that it *accepts*; a ground term $t$ is accepted iff it can be transformed using the transitions as ground rewrite rules to one of the final states. An FTA is a *bottom-up deterministic* (DFTA) if and only if $\Delta$ contains no two transitions with the same left hand side. A (D)FTA is *complete* if for all n-ary functions $f \in \Sigma$ and states $q_1, \ldots, q_n \in Q$, there exists a state $q \in Q$ such that $f(q_1, \ldots, q_n) \to q \in \Delta$. For any FTA there is an equivalent complete DFTA[5]. An optimised algorithm for

* The research leading to these results has been supported by EU FP7 project 318337, *ENTRA* and EU FP7 project 611004, Coordination and Support Action *ICT-Energy*

[5] However in general there is no equivalent top-down deterministic FTA.

determinising and optionally completing a given FTA is described in detail in [7]. The algorithm is developed in a number of stages starting from the textbook algorithm [3]. These stages are (i) minor restructuring; (ii) introduction of functional notation; (iii) modifying the termination condition, and delaying computation of transitions; (iv) separate handling of 0-arity functions; (v) inner loop optimisation enabled by the previous steps; (vi) tracking new values on each iteration; (vii) returning transitions in product form. The final step gives a compact representation of the DFTA. A *product transition* is of the form $f(Q_1, \ldots, Q_n) \rightarrow q$ where $Q_1, \ldots, Q_n$ are sets of states and $q$ is a state. This product transition denotes the set of transitions $\{f(q_1, \ldots, q_n) \rightarrow q \mid q_1 \in Q_1, \ldots, q_n \in Q_n\}$. Thus $\prod_{i=1}^{n} |Q_i|$ transitions are represented by a single product transition.

*Experiments.* We compared extensively the optimised DFTA algorithm with `libvata`[6], an optimised FTA library implemented in C++ (our algorithm is implemented in Java). On a large benchmark set (from `libvata`) of over 14,000 FTA, our procedure was able to determinise and complete over 99% of them within a timeout of 120 seconds, while `libvata` computed the complement of less than 2% of the set with the same timeout. See [7] for more details on the benchmarks. For large randomly generated FTA (using a generator kindly supplied by R. Mayr and R. Almeida), the success rates were 10.9% and 1.4% respectively (that is, random FTA are more difficult for both tools). For FTA inclusion, `libvata` (which does not use determinisation) performs substantially better but the determinisation-based approach is as good for inclusion testing for random FTA, for which inclusion seldom holds.

*Applications.* Determinisation and completion can be used to form the complement of an FTA, leading to applications in verification and analysis using tree automata to represent sets of states, e.g. [5,15,11,6,1]. Gallagher *et al.* [8,9] showed that program properties for analysis of logic program can be formulated as "regular types" defined by an FTA on the program signature; a precise abstract domain for static analysis could then be constructed by determinising the FTA. The approach was made practical by encoding the product transitions directly (using BDDs) thus avoiding the need to enumerate transitions explicitly. DFTA states, which are sets of FTA states, encode useful information in themselves, without generating the DFTA transitions. The emptiness of intersections of input FTA states can be checked by examining the DFTA states. The containment problem for FTA $A_1$ and $A_2$ can also be checked by examining the states of the DFTA obtained from $A_1 \cup A_2$. This has applications, among others, in XML language containment checking. Recently, the determinisation algorithm has been used to implement a refinement procedure for Horn clause verification [14]. An FTA is built to represent the set of derivations in a given set of Horn clauses. Infeasible traces can be eliminated from the FTA by application of the determinisation algorithm to construct the difference of two FTA, which is then used to construct a new set of Horn clauses in which the infeasible traces can no longer be constructed.

[6] http://www.fit.vutbr.cz/research/groups/verifit/tools/libvata/

## References

1. E. Balland, Y. Boichut, P.-E. Moreau, and T. Genet. Towards an efficient implementation of tree automata completion. In *Algebraic Methodology and Software Technology, 12th International Conference, AMAST*, pages 67–82, 2008.
2. W. Charatonik and A. Podelski. Set-based analysis of reactive infinite-state systems. In B. Steffen, editor, *Proceedings of TACAS'98*, volume 1384 of *Springer-Verlag Lecture Notes in Computer Science*, 1998.
3. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.
4. H. Comon, D. Kozen, H. Seidl, and M. Vardi. Applications of Tree Automata in Rewriting, Logic and Programming. Schloß Dagstuhl Seminar 9743, October 20-24, 1997.
5. J. Elgaard, A. Møller, and M. I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proc. Programming Languages and Systems, 9th European Symposium on Programming, ESOP '00*, volume 1782 of *LNCS*, pages 182–194. Springer-Verlag, March/April 2000.
6. G. Feuillade, T. Genet, and V. V. T. Tong. Reachability analysis over term rewriting systems. *J. Autom. Reasoning*, 33(3-4):341–383, 2004.
7. J. P. Gallagher, M. Ajspur, and B. Kafle. An optimised algorithm for determinisation and completion of finite tree automata. *CoRR*, abs/1511.03595, 2015.
8. J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, volume 3132 of *Springer-Verlag Lecture Notes in Computer Science*, pages 27–42, 2004.
9. J. P. Gallagher, K. S. Henriksen, and G. Banda. Techniques for scaling up analyses based on pre-interpretations. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the 21st International Conference on Logic Programming, ICLP'2005*, volume 3668 of *Springer-Verlag Lecture Notes in Computer Science*, pages 280–296, 2005.
10. J. P. Gallagher and G. Puebla. Abstract interpretation over non-deterministic finite tree automata for set-based analysis of logic programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages (PADL'02)*, LNCS, January 2002.
11. T. Genet and V. V. T. Tong. Reachability analysis of term rewriting systems with Timbuk. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 2001.
12. J. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In J. D. P. Rolim, editor, *15 IPDPS 2000 Workshops, Proceedings*, volume 1800 of *Springer-Verlag Lecture Notes in Computer Science*, pages 977–984. Springer, 2000.
13. N. Heintze. Using bottom-up tree automaton to solve definite set constraints. Unpublished. Presentation at Schloß Dagstuhl Seminar 9743, 1997.
14. B. Kafle and J. P. Gallagher. Tree automata-based refinement with application to Horn clause verification. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *VMCAI 2015*, volume 8931 of *Lecture Notes in Computer Science*, pages 209–226. Springer, 2015.
15. D. Monniaux. Abstracting cryptographic protocols with tree automata. *Sci. Comput. Program.*, 47(2-3):177–202, 2003.

# Delta-Oriented FSM Based Testing

(Extended Abstract)*

Mahsa Varshosaz and Mohammad Reza Mousavi

Center for Research on Embedded Systems, Halmstad University, Sweden
{mahsa.varshosaz,m.r.mousavi}@hh.se

Software Product Line (SPL) engineering is a promising approach for software mass development aiming at efficiency in production and customisation. An SPL is a family of similar software systems which share a common set of core assets. In SPL engineering, systematic reuse is enabled in different development phases by exploiting the well-defined commonalities and variabilities among the products of an SPL. Testing is an important and vital part of the software development lifecycle; testing SPLs can be more challenging and costly due to the potential large number of the products. Hence, testing and particularly model-based testing (MBT) of SPLs have become topics of interest in both academia and industry [1, 2, 6]. In this abstract, we briefly explain the steps taken towards adopting a model-based testing technique, namely Harmonized State Identification (HSI) method [3], for testing SPLs. Our approach is built upon the idea of delta-oriented programming (DOP) [4], which is a software engineering technique used for developing SPLs. In DOP, the SPL implementation is divided into developing a core module and a set of delta modules. The core module can be considered as the implementation of a valid product. Each delta module defines a set of changes in the core module. The product implementations can be obtained by applying a set of delta modules to the core module.

In model-based testing, test cases are derived automatically from a test model that describes the expected behavior of the system under test. The HSI method uses finite state machines (FSMs) as test models. Hence, in our approach we define the core module as an FSM and each delta module describes a set of changes in this FSM. To more practically define the type of changes that can be made by applying a delta module to the core model, we consider DeltaJava [5], which is a framework for implementing SPLs. In DeltaJava, an SPL is specified in terms of application of a set of delta modules (addition/removal of object members, methods and classes) to a core product. In the adopted HSI method, DeltaJava is used as a basis for the structure of test models. By giving a semantic domain for DeltaJava programs in terms of FSMs, we motivate the definition of the type of changes that can be made to a core test model, which is an FSM, with respect to the changes made by delta modules in DeltaJava programs.

A finite state machine is a 6-tuple such as $(S, s_0, I, O, \mu, \lambda)$, where $S$ denotes a finite set of states, $s_0$ is the initial state, $I$ is a finite set of input symbols, $O$ is a finite set of output symbols, $\mu : S \times I \to S$ is the transition function, and $\lambda : S \times I \to O$, is the output function. Assuming $s$ to be the current state, by receiving input $i$ the machine moves to state $\mu(s,i)$ and generates the output $\lambda(s,i)$. Assuming that $\sigma$ is a sequence of input symbols, i.e., $\sigma \in I^*$ and $s$ is the current state; for each state such as $s$, we assume $\mu(s,\sigma)$ denotes the state in which the machine ends in after receiving the input symbols in $\sigma$ one by one. Also we consider $\lambda(s,\sigma)$, denotes the sequence of outputs generated by the machine after receiving the input symbols in $\sigma$. Assuming that $X \subseteq I^*$; two states in an FSM are called $X$-equivalent, denoted by $M \equiv_X M'$, if and only if for each sequence $\sigma \in I^*$, the generated output sequence are the same. Considering two FSMs such as $M$ and $M'$, it is said that $M$ and $M'$ are $X$-equivalent, if

*This abstract summarises the main idea of our work published in [7], in addition to its possible extensions.

and only if each state in $M$ has an $X$-equivalent state in $M'$ and vice versa. Lastly, $M$ is said to conform to $M'$ if and only if $M$ and $M'$ are $I^*$-equivalent.

The main idea of the HSI method is to establish the conformance between a test model $M$ and an unknown machine $M'$ that is modeling the behavior of the implementation. There is a set of assumptions made by the HSI method that should hold by $M$ and $M'$ as follows. Both machines are deterministic and minimal, all states in each machine are reachable from the initial state of the machine, and each of the machines has a reliable reset sequence which takes the machine to the initial state from the current state, and finally, $M'$ has at most as many states as $M$. The HSI method, consists of two main phases. In the first phase, the existence of states in the implementation machine which are $I^*$-equivalent with one of the states in the test model is checked. The second phase comprises checking the conformance of the target state and the output of outgoing transitions for corresponding states. The HSI method, uses the following two sets for test case generation. The first set if is called the *state cover set* which is used for reaching states from the initial state. For an FSM $(S, s_0, I, O, \mu, \lambda)$, the state cover set, denoted by $Q$, is a set of input sequences s.t. $\forall_{s \in S} \cdot \ \exists_{q \in Q} \cdot \ \mu(s_0, q) = s$. The *separating family of sequences*, denoted by $Z$, is another set used by the HSI method for generating test cases. This set comprises the sets of separating sequences for all states. Assume that $Pref(.)$ denotes the set of prefixes of a set of sequences, and consider the FSM $(S, s_0, I, O, \mu, \lambda)$; for each state $s \in S$, the set of separating sequences, denoted by $z_s$, is a set of sequences used for distinguishing $s$ from other states in $S$, which is defined as $\forall_{s,s' \in S} \cdot s \neq s' \ \Rightarrow \ \exists_{x \in Pref(z_s) \cap Pref(z_{s'})} \cdot \ \lambda(s, x) \neq \lambda(s', x)$. Hence, the family of separating sequences is defined as $\bigcup_{s \in S} \ \{z_s\}$. The test cases used in the first phase of the HSI method are generated by the concatenation of the sequences in $Q$, and $Z$ and the reset sequence $r$ as follows: $\bigcup_{s \in S} r.q_s.z_s$, where $q_s$ and $z_s$, respectively, denote a sequence in $Q$ which is used to reach $s$ and the set of separating sequences for $s$. In the second phase of the HSI method, the target state and the output of the transitions that are not covered in the first phase while traversing the state cover set are checked.

Using DeltaJava [5], an SPL is specified in terms of a core module and a set of delta modules. The core module represents the implementation of a valid configuration in the SPL which comprises sets of Java classes and interfaces. A delta module describes a set of changes in the core module which can be addition or removal of the fields, methods and also classes and interfaces in the core module. To better define the type of changes in the modelling level, we need to consider the effect of applying the DeltaJava delta modules on the test models. To this end, we define a semantic domain for DeltaJava programs based on FSMs.

The core model, which is an FSM, describes the behavior of a set of objects instantiated from the classes comprised in the core module. The states in the core model, are symbolic states, which correspond to the aggregation of a set of concrete states where each concrete state represents a valuation of variables. The transitions in the test model correspond to the method calls in the objects. Hence, a method call/return is considered as the input/output of a transition. Each method call manipulates a set of variables in the abstract state which results in moving from the current state to the state that comprises the valuation of the variables after performing the method call. Based on the given description, we assume that the set of fields used and manipulated by a method call, its possible return values and its effect on the value of these fields are known. (We assume that the granularity of the abstraction in the test model is modeler's choice, as long as it respects the HSI assumptions). Given the definition of the core model, we can represent the effect of addition/removal of fields by addition/removal of variables to/from the abstract states in the core model. Also, addition/removal of methods can be represented by addition/removal of corresponding transitions from/to the core model. We can describe the behavior of an object in terms of an FSM (with states that correspond to

the abstract evaluations of variables in the object and transitions corresponding to the possible method calls), hence the addition of a class to the set of classes included in the core module, which may result in addition of an object to the set of objects included in the core FSM, can be defined as composition of two FSMs (the FSM describing the behaviour of the object and the core model). Also, the removal of a class, can result in removal of objects instantiated from that class, which results in the removal of the fields and methods corresponding to the object, from the core model. Hence, the result of application of a delta to the core model is another FSM. Since, the resulting FSM should be a valid test model for HSI method (holding the assumptions made by the HSI method), a set of conditions on the way that a delta can change the core model should be defined. (See [7] for more details about the semantic domain that is given for a subset of DeltaJava syntax.)

The main idea of the adopted HSI method is to generate the test cases for each product by reusing the set of test cases generated from the core model, extended by a set of test cases generated by considering the changes made to the core model. To this end, the delta-oriented approach partly reuses the state cover set $Q$ and the family of separating sequences $Z$, generated for the core model. For example by adding methods, some transitions are added to the core model, hence $Q$ can be reused as the state cover set for the resulting FSM. We motivate the efficiency as a criterion for the delta-oriented testing technique. Assuming that the application of the delta modules does not change the size of the core model considerably; the adapted technique is efficient if the total time to generate test cases for all products using the HSI method is much larger than the summation of the time to generate the test cases for the core model using the HSI method and the time to generate test cases for other products by reusing and modifying these test cases using the delta-oriented testing technique. We applied the adopted method (which supports the incremental changes to the core model) to a software system from the health-care domain. We considered the proportion of time required to generate test cases for test models using the delta-oriented approach, and using the plain monolithic HSI method. The results show that the delta-oriented approach can increase the efficiency of test-case generation (up to 50 percent for the considered case study).

## References


[1] E. Engström and P. Runeson. Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, 53(1):2–13, 2011.

[2] S. Oster, A. Wübbeke, G. Engels, and A. Schürr. Model-based software product lines testing survey. In *Model-based Testing for Embedded Systems*, pages 339–381. CRC Press, 2011.

[3] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Comput. Netw. ISDN Syst.*, 15(4):285–297, September 1988.

[4] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. *Delta-Oriented Programming of Software Product Lines*, pages 77–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[5] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, pages 77–91, Berlin, 2010. Springer.

[6] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical report FIN-004-2012, School of Computer Science, University of Magdeburg, Germany, 2012.

[7] M. Varshosaz, H. Beohar, and M. R. Mousavi. Delta-oriented fsm-based testing. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, pages 366–381, 2015.

# Enforcement and Specification of Evolving Privacy Policies for Social Networks

Raúl Pardo

Deptartment of Computer Science and Engineering, Chalmers University of Technology, Sweden.
pardo@chalmers.se

## 1 Motivation

*Online Social Networks* have exploded in popularity in the last years. Over the past decade, the use of Facebook and Twitter, just to mention two of the most popular ones, has increased at the point of becoming ubiquitous. Nearly 70% of the Internet users are active on social networks as shown by a recent survey [12], and this number is increasing. Unfortunately, a number of studies show that the number of privacy breaches is keeping pace with this growth [15, 10, 14, 16]. Very often users' requirements are far from the privacy guarantees offered by social networks which do not meet their expectations. The reasons for that are multifold, ranging from the users' lack of knowledge on the underlying technology to fundamental technical issues of the technology itself. Just to mention a few: i) Many users are not aware of the implications on sharing something in social networks, and do not foresee the consequences until it is too late; ii) The privacy settings offered by existing social networks are limited and are not fine-grained enough to capture desirable privacy policies; iii) Privacy settings are static (they are not time- nor context-dependent), thus not being able to capture the possibility of defining repetitive or recurrent privacy policies.

We are here only concerned with the fact that the privacy settings currently available in social networks are not suitable for capturing the *dynamic* aspect of privacy policies (item iii) in the previous list). That is, privacy policies should take into account that the networks *evolve*, as well as the privacy preferences of the users. The privacy policy may "evolve" due to explicit changes done by the users (e.g., a user may change the audience of an intended post to make it more restrictive), or because the privacy policy is dynamic *per se*. Examples of the latter, are for instance: *"My boss cannot know my location between 20:00-23:59 every day"*, or *"Only my friends can see the pictures I am tagged in from Fridays at 20:00 till Mondays at 08:00"*. These are recurrent policies triggered by some time events ("every day between 20:00 and 23:59", and "every week from Friday at 20:00 till Monday at 08:00"). Other policies may be activated or deactivated by certain events: *"Only up to 3 posts, disclosing my location, are allowed per day in my timeline"*. We call this type of privacy policies *evolving privacy policies*. We have initiated the development of two different formalisms to tackle the problem of evolving privacy policies in social networks. These are: *policy automata* [17] and $\mathcal{TFPPF}$ [18].

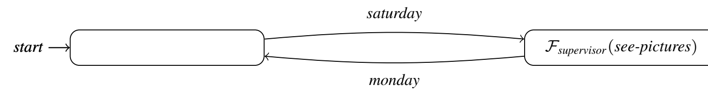
## 2 Approaches to Evolving Privacy Policies

### 2.1 Policy automata

Most social networks include a static privacy policy language in which users can express policies such as "Nobody can see my location" or "Only friends or friends can send me a friend request", i.e., static policies that do not change or evolve as events happen in the system.

In order to describe evolving policies, we take a static policy language for social networks and use it to describe temporal snapshots of the policies in force. We then use an automaton to describe how a policy is discarded and another enforced, depending on the events taking place.

**Example 1.** *Consider the evolving policy "My supervisor cannot see my pictures during the weekend". Let $\mathcal{F}_{supervisor}(\textit{see-pictures})$ represent the static policy "My supervisor cannot see my pictures". The following policy automaton models the policy:*




*The automaton is composed by two states. An initial state in which no static policies need to be enforced and another state in which the policy $\mathcal{F}_{supervisor}(\textit{see-pictures})$ is activated. The transitions are labelled with events reported by the social network. The outgoing transition in the initial state is triggered when the social network reports that Saturday started to the automaton. At this moment, the automaton reports to the social network that the policy needs to be activated. Similarly, when Monday starts and it is communicated to the automaton, it updates again its state and the policy is disabled.*

Policy automata can be converted to *Dynamic Automata with Timers and Events* (DATEs) [3] which are symbolic automata aimed at representing monitors and have the compilation tool LARVA. We implemented the link between LARVA monitors (implementing our policy automata) and the open-source social network Diaspora* [4], thus providing support for some evolving privacy policies [7].

## 2.2 $\mathcal{TFPPF}$

$\mathcal{TFPPF}$ stands for *Temporal First-Order Privacy Policy Framework*. It is a formal framework based on epistemic logic [5] to express and reason about *dynamic* and *recurrent* privacy policies that are activated or deactivated by context (events) or time. We take $\mathcal{FPPF}$ [19] as a point of departure. $\mathcal{FPPF}$ is a formal framework for privacy policies which consists of: i) A generic social network model (SNM); ii) A knowledge-based logic ($\mathcal{KBL}$) to reason about the social network and privacy policies; iii) A formal language ($\mathcal{PPL}$) to describe privacy policies (based on the previous logic). $\mathcal{FPPF}$ is a an expressive privacy policy framework able to represent all privacy policies for social networks like Facebook and Twitter, and beyond [19]. Though rich in what concerns to its expressiveness, $\mathcal{FPPF}$ is not suitable to express evolving privacy policies, in the sense discussed in Section 1. Therefore, $\mathcal{TFPPF}$ extends $\mathcal{FPPF}$ with the following components:

**Traces of SNMs.** In $\mathcal{FPPF}$ we checked whether a policy is in conformance with an SNM, which is a static picture of the system at a moment in time. For evolving policies it is not enough. We need to consider all the SNMs included in the time-frame in which the policy must hold. Therefore, the satisfaction relation for $\mathcal{TKBL}$ (timed $\mathcal{KBL}$) and the conformance relation for $\mathcal{TPPL}$ (timed $\mathcal{PPL}$) both take as a parameter a trace of SNMs. A trace of SNMs is just a sequence of pairs containing an SNM and a timestamp, which indicates the state of the system and the point in time.

**Temporal modalities in $\mathcal{TKBL}$.** The knowledge based logic, $\mathcal{KBL}$, also needs to be extended. $\mathcal{KBL}$ is a first order logic which includes the knowledge modality $K$ from epistemic logic. Using this logic we can reason about what the users know, what are their relationships and what they are permitted to do. However, it was not syntactically possible to specify that a given property always ($\Box$) holds in a trace or eventually will hold ($\Diamond$). These temporal modalities are included in $\mathcal{TKBL}$. Their presence in $\mathcal{TKBL}$ is not optional, since the conformance relation for $\mathcal{TPPL}$ depends on $\mathcal{TKBL}$'s satisfaction relation. In order for a policy to be in conformance with a trace, it is required to hold for all SNMs of the trace corresponding with the time frame specified in the privacy policy.

**Time fields in $\mathcal{TPPL}$.** In $\mathcal{PPL}$ we were able to express all privacy policies of Facebook and Twitter [19], and we conjecture that the language is more expressive that the access control protection present in social networks [2, 6] (though formally showing it is part of our future work). In $\mathcal{TPPL}$, we add a time field where we express the starting time of the privacy policies, i.e., the moment in time from

which it is active; the durantion of the policy, for how long it will be activated, for instance, 6 hours, a day or 10 minutes; and the recurrence of the policy, e.g., monthly, daily, yearly, every other day, etc.

**Example 2.** *Take as an example the policy we modelled using policy automata "My supervisor cannot see my pictures during the weekend". First, let us model the static part of the policy, i.e., "My supervisor cannot see my pictures". We can use the untimed* $\mathcal{PPL}$ *for it,* $[\![\forall \eta . \neg K_{supervisor(me)} picture(\eta)]\!]_{me}$ *where* $\eta \in \mathbb{N}$ *is an index for pictures,* $Ag$ *is the set of all users in the social network,* $me \in Ag$ *is a user and* $supervisor : Ag \to Ag$ *is a function returning the supervisor of a given user.* $K_{supervisor(me)} picture(\eta)$ *reads as "My supervisor knows picture(η)", since the formula is negated and the index for pictures universally quantified we get the desired policy. In order to transform this policy into an evolving one we need to add the temporal part, i.e., "during the weekend".* $\mathcal{TPPL}$ *offers support for it. Assuming that we activate the policy on Saturday 27-08-2016, the policy is written as follows* $[\![\forall \eta . \neg K_{supervisor(me)} picture(\eta)]\!]_{me}^{[\text{ 27-08-2016 } \mid \text{ 2 days } \mid \text{ weekly }]}$*. Concretly, the format of the time field of* $\mathcal{TPPL}$ *privacy policies is* $[start \mid duration \mid recurrence]$*. Thus, the previous must be enforced from 27-08-2016 (Saturday) during 2 days (the weekend) and weekly (every weekend).*

# 3 Current and Future Work

Policy automata and $\mathcal{TFPPF}$ are just the first step towards a complete solution for evolving privacy policies. While these approaches provide support for a whole new family of privacy policies they still have some limitations. For instance, consider that Alice enables the following policy *"Only my friends can see my pictures during the weekend"*. Imagine that Alice and Bob are not friends. If Alice shares a picture on Saturday, Bob will not have access to it. However, on Monday this policy would be deactivated. What would be the effect of turning off this policy? It might be possible that Bob gains access to all the pictures that Alice posted during the weekend, since no restrictions are specified outside the scope of the weekend. In order to address this problem we might need a policy language able to express *real-time* aspects. For $\mathcal{TFPPF}$ we are currently extending the logic with timestamps in predicates and modalities. It gives the needed expressive power to protect some piece of information shared at a given moment in time. For policy automata it is bit more complicated, since the approach is policy language agnostic. One way for policy automata to solve this problem is being aware of the timestamp of the pieces of information it is protecting (or monitoring). Combining the real-time extension of $\mathcal{TFPPF}$ and policy automata might result in a feasible solution. We are studying how to adapt policy automata to a policy language that contains timestamps indicating when some information has been shared.

Furthermore, the connection between policy automata and $\mathcal{TFPPF}$ is an interesting question even in their current form. $\mathcal{TFPPF}$ does not have an enforcement mechanism, so it can be seen as a language suitable for reasoning and specification of privacy policies. On the other hand, policy automata are enforceable through LARVA monitors, but it is not so targeted for policy reasoning. We plan to investigate what is the connection between the two formalisms and how to convert $\mathcal{TFPPF}$ policies into policy automata, thus providing an enforcement mechanism for $\mathcal{TFPPF}$.

**Related work.** To the best of our knowledge we are the first to offer support for specification and enforcement of evolving privacy policies. However, the techniques we use has extensively been used in the security literature. Epistemic logic has been used in the form of interpreted systems (which include temporal operators as the ones used in $\mathcal{TFPPF}$) to describe access control and information flow properties in multiagent systems [5, 9, 1, 20]. Moreover, we are currently showing the correspondance between traditional Kripke semantics and the semantics based on SNMs for $\mathcal{FPPF}$. As for policy automata, there is work done in the context of security policies, for instance the work by Le Guernic *et al.* on using automata to monitor and enforce non-interference [11, 8] or by Schneider on security automata [21]. It could be instructive to further develop the theoretical foundations of policy automata and relate it to security automata and their successors (e.g., edit automata [13]).

## References


[1] M. Balliu. A logic for information flow analysis of distributed programs. In *Secure IT Systems*, pages 84–99. Springer, 2013.

[2] G. Bruns, P. W. Fong, I. Siahaan, and M. Huth. Relationship-based access control: its expression and enforcement through hybrid logic. In *CODASPY'12*, pages 117–124. ACM, 2012.

[3] C. Colombo, G. J. Pace, and G. Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *13th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'08)*, volume 5596 of *LNCS*, pages 135–149, L'Aquila, Italy, September 2009. Springer-Verlag.

[4] Diaspora*. https://diasporafoundation.org/. Accessed: 2016-08-27.

[5] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about knowledge*, volume 4. MIT press Cambridge, 1995.

[6] P. W. Fong. Relationship-based access control: Protection model and policy language. In *CODASPY'11*, pages 191–202. ACM, 2011.

[7] $\mathcal{PPF}$ Diaspora*. Test pod: https://ppf-diaspora.raulpardo.org. Code: https://github.com/ raulpardo/ppf-diaspora. 2016.

[8] G. L. Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 218–232, 2007.

[9] J. Y. Halpern and K. R. O'Neill. Secrecy in multiagent systems. *ACM Transactions on Information and System Security (TISSEC)*, 12(1):5, 2008.

[10] M. Johnson, S. Egelman, and S. M. Bellovin. Facebook and privacy: It's complicated. SOUPS '12, pages 9:1–9:15, New York, NY, USA, 2012. ACM.

[11] G. Le Guernic, A. Banerjee, T. Jensen, and D. A. Schmidt. *Automata-Based Confidentiality Monitoring*, pages 75–89. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[12] A. Lenhart, K. Purcell, A. Smith, and K. Zickuhr. Social media & mobile internet use among teens and young adults. millennials. *Pew Internet & American Life Project*, 2010.

[13] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4:2–16, 2005.

[14] Y. Liu, K. P. Gummadi, B. Krishnamurthy, and A. Mislove. Analyzing facebook privacy settings: User expectations vs. reality. IMC '11, pages 61–70. ACM, 2011.

[15] M. Madejski, M. Johnson, and S. Bellovin. A study of privacy settings errors in an online social network. PERCOM Workshops'12, pages 340–345. IEEE, 2012.

[16] M. Madejski, M. L. Johnson, and S. M. Bellovin. The failure of online social network privacy settings. 2011.

[17] R. Pardo, C. Colombo, G. J. Pace, and G. Schneider. An automata-based approach to evolving privacy policies for social networks. In *RV'16*, LNCS, 2016. To appear.

[18] R. Pardo, I. Kellyérová, C. Sánchez, and G. Schneider. Specification of evolving privacy policies for online social networks. In *TIME'16*. IEEE Society Press, 2016. To appear.

[19] R. Pardo and G. Schneider. A formal privacy policy framework for social networks. In *SEFM'14*, volume 8702 of *LNCS*, pages 378–392. Springer, 2014.

[20] J. Ruan and M. Thielscher. A logic for knowledge flow in social networks. In *AI 2011: Advances in Artificial Intelligence*, pages 511–520. Springer, 2011.

[21] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

# Non-interleaving opperational semantics for late and early Pi-calculus

Håkon Normann*

Section of Theoretical Computer Science, IT-University of Copenhagen,
Rued Langgaardsvej 7, 2300 Copenhagen, Denmark

**Abstract**

We provide a new non-interleaving operational semantics for the late and early Pi-calculus based on the classical model of Asynchronous Transition Systems (ATS). Our aim is to give a semantic that is as close as possible to the original early and late pi-calculus semantic, while the transition system generated is rich enough that we can define an independence relation on it to provide non-interleaving. This we aim by extending the original semantics with a notion of term location, and extruder histories. This added with the idea of splitting events we generate an ATS for both the early and late pi-calculus.

## 1 Introduction

In this work we report on a new non-interleaving operational semantics for the Pi-calculus [MPW92].

The semantics is developed as part of a PhD project affiliated to the Computational Artifacts (CompArt) research project [1] suported by the Velux foundation. The project we focuses on extending the mathematical foundations of computational coordination technologies, with particular focus on the ability to express causality and adaptability in systems of communicating processes.

The pi-calculus [MPW92] is a seminal foundational model for communicating systems with dynamic communication topology. A *dynamic communication topology* means that new connections for communication can be established between concurrent processes during an ongoing computation. Dynamic connections can model mobility of processes, by representing the location of a process by which other processes it is connected to. This notion of location however completely abstracts from concurrency: Indeed, the pi-calculus was initially given a so-called *interleaving* semantics in terms of transition systems [MPW92]. An interleaving semantics abstracts from concurrency and causality, representing two parallel computations as all the possible sequential interleavings, and thus does not distinguish between processes like $a.b + b.a$ and $a \,||\, b$ in the transition system.

Other notions of location are relevant in describing communication processes, in particular when it comes to causality and adaptability. The fact that two processes are running concurrently, that is, independently at different physical locations, has influence on the behaviour of an adapted system, where only one of the processes has been changed. If one for instance refines [vGG01] the sub process $b$ above with the sub process $c.d$ the resulting two processes $a.c.d + c.d.a$ and $a \,||\, c.d$ will not have the same interleaving behaviour, since the latter will have an additional interleaving $c.a.d$.

Moreover, an action in one of two parallel processes can only be causally related to an action in the other, if there is some communication between the two processes in between the actions. This is relevant in the study of *reversible computing*, including the reversible

*I would like to thank my supervisors Thomas Hildebrandt and Christian Johansen for their help.

[1] Compart.ku.dk

Pi-calculus [CKV13]. The reason is, that if two actions $a$ and $b$ happen concurrently and independently as in $a \,||\, b$, and computation is reversed, both $a$ and $b$ should be able to be the first action to be un-done, no matter in which order they were observed.

These observations motivate research in so-called non-interleaving semantics for concurrent process calculi using semantic models able to express concurrency explicitly, such as Petri Nets, event structures [WN95, NPW81, Win86, Sec.10] or asynchronous transition systems [Bed88, Shi85].

Recent work by Crafa, Varacca, and Yoshida [CVY12, Sec.6] provides *denotational* non-interleaving semantics for the pi-calculus and the work on reversible pi [CKV13] provides as bi-product an operational non-interleaving semantics for *finite* pi-calculus processes without branching. Both lines of work only consider the *late* semantics of pi-calculus.

We are interested in a non-interleaving, operational semantics for both late and early pi-calculus that allows finite representations of processes with infinite behaviour. The finite systems semantics will enable static analysis and model-checking techniques not available for infinitary semantics. Still, we want to relate our semantics to existing denotational semantics provided using e.g. the model of event structures.

We obtain this by generalising the approach by Mukund and Nielsen [MN92] in giving a non-interleaving operational semantics for CCS [Mil80] as labelled Asynchronous Transition Systems (ATS) [Bed88, Shi85] that generalizes transition systems and can also be unfolded as event structures [WN95, Sec.10]. The work of Mukund and Nielsen captures the *structural* (or subjective) causality which comes from the structure of a process, e.g. sequencing and parallel composition, using a notion of transition location inferred from the structure of the terms.

We capture the additional so-called *link* (or objective) causalities introduced in pi-calculus processes by taking inspiration from the work by Crafa, Varacca, and Yoshida [CVY12, Sec.6]. This causality is the one that comes from the communication of names, specially one can see it in the example $(\nu n)(\overline{a}\langle n\rangle \,||\, \overline{b}\langle n\rangle \,||\, \underline{n}(x))$ where the input on $n$ is depending on the *first* of the outputs on $a$ and $b$ to extrude $n$ outside of the viewed scope, but it should be independent on the *second* output of the name $n$. We also should assume that the two outputs should be mutually independent. This is sometimes called *disjunctive causality* and cannot be captured by prime event structures. Compared to the work in [CVY12, Sec.6], the main differences of our work are: (i) we provide an operational semantics, (ii) we capture both late and early semantics, and (iii) we split events based on so-called extrusion histories and thereby obtain a stable non-interleaving semantics that unfolds to standard prime event structures (i.e., splitting is the way to model disjoint causalities in a stable event-based concurrency model).

Our semantics has relationships to the semantics of reversible pi-calculus [CKV13] (Rpi), which provides a stable causal semantics as a bi-product. Concretely, the memories in Rpi carry the information of extrusion that we record in histories, but carry also more more information (needed for the purpose of reversibility). Also, the semantics of Rpi in [CKV13] only covers finite processes without branching. We believe that our work shows that branching (choice) and replication introduce new subtleties in the causal history which is not just a straightforward generalisation of the semantics for finite, non-branching processes. Moreover, the semantics in [CKV13] does not explicitly provide the events and thus neither the asynchronous transition system underlying the semantics.

## 2 On early vs late semantics

In the early semantics of pi-calculus when considering an input prefix, such as $\underline{a}(x).P$, it is determined already in the input rule the exact name $n$ that is received, thus replacing all

occurrences of $x$ in $P$. This gives the possibility to model that we receive a message from outside the modelled process and know what it is. In the transition system it will give one transition for each possible name that can be received, one per non-scoped name appearing in the process and one for all possible free names. The late semantics, however, only determines what is received at the moment of the synchronization between the output and the input takes place. For only inputs, i.e., in the input rule, all we know is that we received something but not really what this is.

For the non-interleaving semantics these differences have some impacts. In the early semantics we may extrude a name that a prefix later receives. For instance, in the process $(\nu n)(\overline{a}\langle n\rangle \,||\, \underline{b}(x))$ the input on channel $b$ may receive the name $n$ previously extruded via channel $a$ – but only after it has been extruded. Thus, in the early semantics we will have different events depending on whether a fresh name is received on channel $b$ (which can happen independently of the parallel output on channel $a$) or if it is the name $n$ that is received. For the late semantics there is no way that a name could be extruded and later received.

## References

[Bed88] Marek A. Bednarczyk. *Categories of asynchronous systems.* PhD thesis, Univ. Sussex, 1988.

[CKV13] Ioana Cristescu, Jean Krivine, and Daniele Varacca. A compositional semantics for the reversible pi-calculus. In *ACM/IEEE Symposium on Logic in Computer Science, LICS*, pages 388–397. IEEE Computer Society, 2013.

[CVY12] Silvia Crafa, Daniele Varacca, and Nobuko Yoshida. Event structure semantics of parallel extrusion in the pi-calculus. In Lars Birkedal, editor, *FOSSACS*, volume 7213 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2012.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92. Springer-Verlag, 1980.

[MN92] Madhavan Mukund and Mogens Nielsen. CCS, Location and Asynchronous Transition Systems. In *FSTTCS*, volume 652 of *LNCS*, pages 328–341. Springer, 1992.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts i and ii. *Information and Computation*, 100(1):1–40,41–77, 1992.

[NPW81] Nielsen, Plotkin, and Winksel. Petri nets, event structures and domains, part I. *TCS: Theoretical Computer Science*, 13:85–108, 1981.

[Shi85] M. W. Shields. Concurrent machines. *Computer Journal*, 28(5):449–465, 1985.

[vGG01] Rob van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001.

[Win86] Glynn Winskel. Event structures. In *Advances in Petri Nets*, volume 255 of *LNCS*, pages 325–392. Springer, 1986.

[WN95] Glynn Winskel and Mogens Nielsen. Models for concurrency. In S. Abramski, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science – vol 4*, pages 1–148. Oxford Univ. Press, 1995.

# Minimizing Markov Chains Beyond Bisimilarity

Giovanni Bacci, Giorgio Bacci, Kim G. Larsen, and Radu Mardare

Department of Computer Science, Aalborg University, Denmark
{giovbacci,grbacci,kgl,mardare}@cs.aau.dk

Probabilistic bisimulation for Markov chains (MCs), introduced by Larsen and Skou [19], is widely considered a key concept for reasoning about the equivalence of probabilistic systems. This concept is intimately related to that of lumpability by Kemeny and Snell [15] and has been efficiently applied in state space reduction techniques for probabilistic systems [7]. Computing the bisimilarity quotient of a Markov chain (a.k.a. the optimal state-space lumping) yields *the smallest* Markov chain that behaves exactly as the original chain. The optimal lumping quotient has a (typically much) smaller state space, nevertheless, it may be still too large to yield significant computational improvements. This is a well known problem that state-of-the-art probabilistic model-checking tools [18, 14, 12] strive to overcome.

The main reason for this phenomenon arises from the fact that probabilistic bisimilarity is a notion of equivalence that is too "exact" for many purposes as it only relates processes with identical bahaviors. In a number of applications, such as systems biology [4], games [5] and machine learning [13], one is interested in knowing whether two probabilistic systems that may differ by a small amount in the real-valued parameters have "sufficiently" similar behaviours.

This motivated the study of behavioral metrics for Markov chains, initiated by Desharnais et al. [8] and greately developed in the last decade [20, 6, 1, 2]. The idea consists in using distances (more precisely, 1-bounded pseudometrics) to measure the behavioral dissimilarities of two MCs, that is, the farther two states are the more distinguishable their behaviors are. In this respect, bisimilar states are at distance zero with each other, while two states that have no significant similarity are at distance 1.

Analogously, also the concept of lumpability has been relaxed by considering partitioning of the state space whose elements are at most $\varepsilon$-apart from each other [9]. Thus, by suitably choosing the value of $\varepsilon$, one can significantly reduce the size of the resulting lumping at the cost of loosing in precision. However, the value of $\varepsilon$ only bounds the relative distance between two given states in the original MC, whereas the distance between the original MC and its $\varepsilon$-lumping is (typically much) greater than $\varepsilon$. Generally, it is difficult to obtain a good tradeoff between the size and the precision of the resulting $\varepsilon$-lumping, and that often requires ad hoc adjustments based on a deep knowledge of the problem domain where the original MC originates.

These issues were also pointed out in [9], leaving open the following problem. Given a finite MC $\mathcal{M}$ and a positive integer $k$, what is its "best" $k$-state approximation? Here by "best" we mean a $k$-state MC of minimal distance to the original. In this paper we address the above question, that we call *the Markov chain Approximation problem* (MCA). The MCA problem is defined as the following optimization problem

$$\min \{\delta(\mathcal{M},\mathcal{N}) \mid \mathcal{N} \in \mathbb{F}(k)\} \,, \qquad \text{(MCA PROBLEM)}$$

where $\mathbb{F}(k)$ denotes the class of all the MCs with at most $k \in \mathbb{N}$ states. If an instance of the MCA problem admits an optimal solution, say $\mathcal{N}^*$, it represents a (suitably small) MC that best approximates the behavior of $\mathcal{M}$, and the *optimal value* $\delta(\mathcal{M},\mathcal{N}^*)$ certifies its quality.

Obviously, for values of $k$ greater than or equal to the cardinality of the bisimilarity quotient, the MCA problem reduces to constructing the optimal lumping quotient of $\mathcal{M}$ which can be solved in $O(n \log m)$ where $n$ is the number of states and $m$ is the number of transitions [7].

As one may expect, the MCA problem is in general non-trivial to solve, moreover it seems unlikely that it can be solved efficiently. We formalize this conjecture by studying the computational complexity of its corresponding decision problem, namely, the *the Threshold MCA problem*. The Threshold MCA problem is defined as follows. Given $k \in \mathbb{N}$, a finite MC $\mathcal{M}$ and $\varepsilon \in \mathbb{Q} \cap [0, 1]$, decide whether $\min \{\delta(\mathcal{M}, \mathcal{N}) \mid \mathcal{N} \in \mathbb{F}(k)\} \leq \varepsilon$.

**Theorem 1.** *The Threshold MCA problem is NP-hard.*

The above complexity lower-bound follows from a stronger result on a specialized version of the Threshold MCA problem, called *the Qualitative MCA problem*.

Formally, the Qualitative MCA problem is defined as follows. Given $k \in \mathbb{N}$ and a finite MC $\mathcal{M}$, decide whether $\min \{\delta(\mathcal{M}, \mathcal{N}) \mid \mathcal{N} \in \mathbb{F}(k)\} < 1$. Recall that, two MCs are at distance 1 from each other when no significant similarity can be observed between their behaviors. In this respect, the Qualitative MCA problem corresponds to the question "Is it worth searching for a $k$-states approximation of $\mathcal{M}$?".

**Theorem 2.** *The Qualitative MCA problem is NP-complete.*

The NP-hardness of the Qualitative MCA problem is obtained by means of a polynomial-time many-one reduction from *vertex cover*. Interestingly, the same reduction can be easily adapted to prove the NP-hardness of the Thresold MCA problem w.r.t. *the total variation distance* (a.k.a. trace distance) or w.r.t. any multi-step bisimilarity distance as defined in [2].

Theorem 2 provides tight complexity bounds for the Qualitative MCA problem, however, for generic error bound $\varepsilon$, the Threshold MCA problem appears to be much harder to solve.

**Theorem 3.** *The Threshold MCA problem is in PSPACE.*

The above complexity upper-bound has been obtained by reducing the Threshold MCA problem to checking the validity of a formula in the existential theory of reals [3].

Notably, our reduction involves only conjunctions of bilinear constraints, therefore their feasibility may be more efficiently checked viewing them as bilinear matrix inequalities (BMIs) for which general algorithms [11, 10] and tools [17, 16] have been developed.

The fact that the MCA problem can be reduced to an optimization problem with bilinear matrix inequalities makes us hope for the possibility to adapt well studied non linear optimization methods to efficiently construct good locally optimal solutions for the MCA problem.

# References

[1] Giorgio Bacci, Giovanni Bacci, Kim G. Larsen, and Radu Mardare. On-the-Fly Exact Computation of Bisimilarity Distances. In *TACAS*, volume 7795 of *LNCS*, pages 1–15, 2013.

[2] Giorgio Bacci, Giovanni Bacci, Kim G. Larsen, and Radu Mardare. Converging from Branching to Linear Metrics on Markov Chains. In *ICTAC*, volume 9399 of *LNCS*, pages 349–367. Springer, 2015.

[3] John F. Canny. Some Algebraic and Geometric Computations in PSPACE. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC'88)*, pages 460–467. ACM, 1988.

[4] Luca Cardelli and Attila Csikász-Nagy. The Cell Cycle Switch Computes Approximate Majority. *Scientific Reports*, 2, 2012.

[5] Krishnendu Chatterjee, Luca de Alfaro, Rupak Majumdar, and Vishwanath Raman. Algorithms for Game Metrics. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008*, volume 2 of *LIPIcs*, pages 107–118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

[6] Di Chen, Franck van Breugel, and James Worrell. On the Complexity of Computing Probabilistic Bisimilarity. In *FoSSaCS*, volume 7213 of *LNCS*, pages 437–451. Springer, 2012.

[7] Salem Derisavi, Holger Hermanns, and William H. Sanders. Optimal state-space lumping in Markov chains. *Inf. Process. Lett.*, 87(6):309–315, 2003.

[8] Josee Desharnais, Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Metrics for labelled Markov processes. *Theoretical Compututer Science*, 318(3):323–354, 2004.

[9] Norm Ferns, Prakash Panangaden, and Doina Precup. Metrics for finite Markov Decision Processes. In *UAI*, pages 162–169. AUAI Press, 2004.

[10] Mituhiro Fukuda and Masakazu Kojima. Branch-and-cut algorithms for the bilinear matrix inequality eigenvalue problem. *Comp. Opt. and Appl.*, 19(1):79–105, 2001.

[11] Keat-Choon Goh, Michael G. Safonov, and George P. Papavassilopoulos. Global optimization for the Biaffine Matrix Inequality problem. *J. Global Optimization*, 7(4):365–380, 1995.

[12] Ernst Moritz Hahn, Yi Li, Sven Schewe, Andrea Turrini, and Lijun Zhang. IscasMC: A web-based probabilistic model checker. In *Nineteenth international symposium of the Formal Methods Europe association (FM)*, volume 8442 of *LNCS*, pages 312–317. Springer, 2014.

[13] Manfred Jaeger, Hua Mao, Kim Guldstrand Larsen, and Radu Mardare. Continuity properties of distances for markov processes. In *Quantitative Evaluation of Systems - 11th International Conference, QEST 2014*, volume 8657 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2014.

[14] Joost-Pieter Katoen, Maneesh Khattri, and Ivan S. Zapreev. A Markov Reward Model Checker. In *Second International Conference on the Quantitative Evaluation of Systems (QEST 2005)*, pages 243–244. IEEE Computer Society, 2005.

[15] John G. Kemeny and James L. Snell. *Finite Markov chains*. Undergraduate texts in mathematics. Springer, New York, 1976. Reprint of the 1960 ed. published by Van Nostrand, Princeton, N.J., in the University series in undergraduate mathematics.

[16] Michal Kočvara and Michael Stingl. PENBMI 2.0. http://www.penopt.com/penbmi.html. Accessed: 2016-08-28.

[17] Michal Kočvara and Michael Stingl. PENNON: A code for convex nonlinear and semidefinite programming. *Optimization Methods and Software*, 18(3):317–333, 2003.

[18] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[19] Kim Guldstrand Larsen and Arne Skou. Bisimulation through probabilistic testing. *Information and Computation*, 94(1):1–28, 1991.

[20] Franck van Breugel, Babita Sharma, and James Worrell. Approximating a Behavioural Pseudometric without Discount for Probabilistic Systems. *Logical Methods in Computer Science*, 4(2), 2008.

# WNetKAT: Programming and Verifying Weighted Software-Defined Networks*

Kim G. Larsen, Stefan Schmid, Bingtian Xue

Aalborg University, Denmark
{kgl,schmiste,bingt}@cs.aau.dk

**Abstract**

Programmability and verifiability lie at the heart of the Software-Defined Networking (SDN) paradigm. This paper presents *WNetKAT*, a network programming language accounting for the fact that networks are inherently weighted, and communications subject to capacity constraints (e.g., in terms of bandwidth) and costs (e.g., latency or monetary costs). *WNetKAT* is based on a syntactic and semantic extension of the NetKAT algebra, and comes with releavant applications, including cost- and capacity-aware reachability testing or service chaining. This paper also initiates the discussion of decidability and the relationship to weighted finite automata.

## 1 WNetKAT

On a high level, a computer network can be described as a set of nodes (hosts or routers) which are interconnected by a set of links, hence defining the network topology. While this high-level view is sufficient for many purposes, for example for reasoning about reachability, in practice, the situation is often more complex: both nodes and links come with capacity constraints (e.g., in terms of buffers, CPU, and bandwidth) and may be attributed with costs (e.g., monetary or in terms of performance). To reason about performance, cost, and fairness aspects, it is therefore important to take these dimensions into account.

The challenge of extending the state-of-the-art SDN language NetKAT [2] to weighted scenarios lies in the fact that in a weighted network, traffic flows can no longer be considered independently, but they may *interfere*: their packets compete for the shared resource. Moreover, packets of a given flow may not necessarily be propagated along a unique path, but may be split and distributed among multiple paths (in the so-called *multi-path routing* or *splittable flow* variant). Accordingly, a weighted extension of NetKAT must be able to deal with "inter-packet states".

We can think of the network as a weighted (directed) graph $G = (V, E, w)$. Here, $V$ denotes the set of switches (or equivalently routers, and henceforth often simply called nodes), $E$ is the set of links (connected to the switches by *ports*), and $w$ is a weight function. The weight function $w$ applies to both nodes $V$ as well as links $E$. Moreover, a node and a link may be characterized by *a vector of weights* and also combine *multiple resources*: for example, a list of capacities (e.g., CPU and memory on nodes, or bandwidth on links) and a list of costs (e.g., performance, energy, or monetary costs).

We propose a weighted extension of NetKAT:

- *WNetKAT* includes a set of *quantitative packet-variables* to specify the quantitative information carried in the packet, in addition to the regular (non-quantitative) packet-variables of NetKAT (called *fields* in NetKAT): e.g., regular variables are used to describe locations, such as switch and port, or priorities, while quantitative variables are used to specify latency or energy. The set of all packet-variables is denoted by $\mathcal{V}_p$.

*Research supported by the Danish VILLUM FONDEN project *ReNet*. A longer version of this paper is available as a technical report on arXiv [4]. We would like to thank Alexandra Silva, Nate Foster, and Dexter Kozen for many inputs and discussions on WNetKAT.

$$[\![x \leftarrow \omega]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk[\omega/x] :: h\} & \text{if } x \in \mathcal{V}_p \\ \{\rho(v)[\omega/x],\ pk :: h\} & \text{if } x \in \mathcal{V}_s \text{ and } pk(sw) = v \end{cases} \quad (1)$$

$$[\![x = \omega]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk :: h\} & \text{if } x \in \mathcal{V}_p \text{ and } pk(x) = \omega \\ & \text{ or if } x \in \mathcal{V}_s, pk(sw) = v \text{ and } \rho(v, x) = \omega \\ \emptyset & \text{otherwise} \end{cases} \quad (2)$$

$$[\![y \leftarrow (\Sigma_{y' \in \mathcal{V}'} y' + r)]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk[r'/x] :: h\} & \text{if } x \in \mathcal{V}_p \\ \{\rho(v)[r'/x],\ pk :: h\} & \text{if } x \in \mathcal{V}_s \text{ and } pk(sw) = v \end{cases} \quad (3)$$
$$\text{where } r' = \Sigma_{y_p \in \mathcal{V}' \cap \mathcal{V}_p} pk(y_p) + \Sigma_{y_s \in \mathcal{V}' \cap \mathcal{V}_q} \rho(v, y_s) + r$$

$$[\![y = (\Sigma_{y' \in \mathcal{V}'} y' + r)]\!](\rho,\ pk :: h) = \begin{cases} \{\rho,\ pk :: h\} & \text{if } x \in \mathcal{V}_p \text{ and } pk(x) = r' \\ & \text{ or } x \in \mathcal{V}_s, pk(sw) = v \text{ and } \rho(v, x) = r' \\ \emptyset & \text{otherwise} \end{cases} \quad (4)$$
$$\text{where } r' = \Sigma_{y_p \in \mathcal{V}' \cap \mathcal{V}_p} pk(y_p) + \Sigma_{y_s \in \mathcal{V}' \cap \mathcal{V}_q} \rho(v, y_s) + r$$

Table 1: Semantics of *WNetKAT*: (1) and (2) describe the semantics for the non-quantitative variable assignment and test respectively. When the variable is a packet variable, the semantics are defined like in NetKAT. When the variable is a switch variable, the semantics of assignment changes the value of the variable in $\rho$ rather of the head packet, and the test compares the value of the variable in $\rho$. (3) and (4) are the semantics for quantitative variable assignment and test respectively. In contrast to the non-quantitative variables, here the values of the variables are naturals and the value for updating the head packet or $\rho$ need to be calculated for the related values.

- *WNetKAT* also includes a set of *switch-variables*, denoted by $\mathcal{V}_s$, to specify the configurations at the switch. Switch variables can either be quantitative (e.g., counters, meters, meta-rules [1, 5]) or non-quantitative (e.g., location related), as it is the case of the packet-variables.

In addition to introducing quantitative variables, we also need to extend the atomic actions and tests of NetKAT. Concretely, *WNetKAT* first supports non-quantitative assignments and non-quantitative tests on the non-quantitative switch-variables, similar to those on the packet-variables in NetKAT. Moreover, *WNetKAT* also allows for *quantitative assignments* and *quantitative tests*, defined as follows, where $x \in \mathcal{V}_q$, $\mathcal{V}' \subseteq \mathcal{V}_q$, $\delta \in \mathbb{N}$, $\bowtie \in \{>, <, \leq, \geq, =\}$:

- **Quantitative Assignment** $x \leftarrow (\Sigma_{x' \in \mathcal{V}'} x' + \delta)$: Read the current values of the variables in $\mathcal{V}'$ and add them to $\delta$, then assign this result to $x$.
- **Quantitative Test** $x \bowtie (\Sigma_{x' \in \mathcal{V}'} x' + \delta)$: Read the current value of the variables in $\mathcal{V}'$ and add them to $\delta$, then compare this result to the current value of $x$.

Given the set of switches $V$, a *switch-variable valuation* is a partial function $\rho : V \times \mathcal{V}_s \hookrightarrow \mathbb{N} \cup \Omega$. It associates, for each switch and each switch-variable, a integer or a value from $\Omega$. We emphasize that $\rho$ is a partial function, as some variables may not be defined at some switches.

A *WNetKAT*expression denotes a function $[\![\ ]\!] : \rho \times H \to 2^H$, where $H$ is the set of packet histories. The semantics of *WNetKAT* is defined in Table 1, where $x \in \mathcal{V}_n, y \in \mathcal{V}_q$, $\delta \in \mathbb{N}$ and $\omega \in \Omega$.

**Example 1.** *Consider the network in Figure 1. The topology of the network can be characterized with the following* WNetKAT *formula $t$, where $sw$ specifies the current location (switch) of the packet, $co$ specifies the cost, and $ca$ specifies the capacity along the links.*

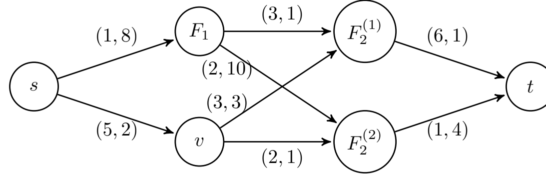



Figure 1: *Example:* A network hosting two (virtualized) functions $F_1$ and $F_2$. Function $F_2$ is allocated twice. The functions $F_1$ and $F_2$ may change the traffic rate.

$$
\begin{aligned}
t ::= \quad & sw = s; (sw \leftarrow F_1; co \leftarrow co + 1; ca \leftarrow \min\{ca, 8\} \\
& \quad \& \; sw \leftarrow v; co \leftarrow co + 5; ca \leftarrow \min\{ca, 2\}) \\
& \& \; sw = F_1; \\
& \quad (sw \leftarrow F_2^{(1)}; co \leftarrow co + 3; ca \leftarrow \min\{ca, 1\} \\
& \quad \& \; sw \leftarrow F_2^{(2)}; co \leftarrow co + 2; ca \leftarrow \min\{ca, 10\}) \\
& \& \; sw = v; (sw \leftarrow F_2^{(1)}; co \leftarrow co + 3; ca \leftarrow \min\{ca, 3\} \\
& \quad \& \; sw \leftarrow F_2^{(2)}; co \leftarrow co + 2; ca \leftarrow \min\{ca, 1\}) \\
& \& \; sw = F_2^{(1)}; sw \leftarrow t; co \leftarrow co + 6; ca \leftarrow \min\{ca, 1\} \\
& \& \; sw = F_2^{(2)}; sw \leftarrow t; co \leftarrow co + 1; ca \leftarrow \min\{ca, 4\}
\end{aligned}
$$

*The variable $co$ accumulates the costs along the path, and the variable $ca$ records the smallest capacity along the path. Notice that $ca$ is just a packet-variable used to record the capacity of the path; it does not represent the capacity used by this packet (the latter is assumed to be negligible).*

*Assume that function $F_1$ is flow conserving (e.g., a NAT), while $F_2$ increases the flow rate by an additive constant $\gamma \in \mathbb{N}$ (e.g., a security related function, adding a watermark or an IPSec header). The policy of $F_2$ can be specified as:* $p_{F_2} ::= \quad (sw = F_2^{(1)} \;\&\; sw = F_2^{(2)}); ca \leftarrow ca + \gamma$

In our future work, we will investigate the decidability of WNetKAT. In particular, it can be seen that a weighted NetKAT automata is a finite state weighted automaton $A = (S, s, F, \lambda, \mu)$ over a structure $K$ and alphabet $\Sigma$. From the equivalence between WNetKAT and weighted automata, it follows that deciding equivalence of two WNetKAT expressions is impossible. However, we also observe that in many practical scenarios, the above undecidability result is too general and does not apply. For example, many practical applications such as cost reachibility can actually be reduced to test *emptiness*: we often want to test whether a given WNetKAT expression $e$ equals $0$, i.e., whether the corresponding weighted NetKAT automaton is empty.

Finally, we note that we currently witness a trend toward computationally more advanced and stateful packet-processing functionality, e.g., in the context of P4 [1] or OpenState [3]: these platforms are hence interesting compilation targets for *WNetKAT*.

For more details, we refer the reader to the accompanying arXiv report [4].

## References

[1] Bosshart et al. P4: Programming protocol-independent packet processors. *SIGCOMM CCR*, 2014.

[2] C. Anderson et al. Netkat: Semantic foundations for networks. *SIGPLAN Not.*, 49(1), January 2014.

[3] G. Bianchi et al. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM CCR*, 2014.

[4] K. Larsen et al. Wnetkat: A weighted sdn programming and verification language. *arXiv*, 2016.

[5] L. Schiff et al. In-band synchronization for distributed sdn control planes. *SIGCOMM CCR*, 2016.

# Testing Delta-oriented Software Product Lines using Differential Symbolic Execution

Sebastian Kunze and Mohammad Reza Mousavi

Halmstad University, Centre for Research on Embedded Systems, Halmstad, Sweden
{sebastian.kunze,m.r.mousavi}@hh.se

## 1 Introduction

*Software Product Line Engineering* (SPLE) facilitates managing a family of software products that share certain commonalities and differentiate in specific variability points. Several companies and institutions have successfully applied SPLE to structure the development of embedded and safety-critical systems for reuse.[1] The main advantage of SPLE is in generating a group of customised products with a reduced development costs and shorter time to market. However, at the same time, it gives rise to a complex structure and interactions among the features involved in those products. Thus, verification and validation of a Software Product Line (SPL) becomes challenging.

*Delta-oriented Programming* (DOP) [4] is an approach to SPLE that enables reuse of certain program artefacts. Those artefacts are specified within *delta modules* that allow for adding to, removing from, or modifying particular features of a *core module* to derive customised products. Compared to other approaches to SPLE, DOP supports flexible design and modular evolution.

Although testing a safety-critical SPL in a rigorous and thorough manner is essential, it is rarely done properly because of the number of product configurations based on the variability points. To address the issue of efficiency and effectiveness in SPLs, several analysis approaches have been proposed and developed in the past; we refer to Thüm et al. [5] for a detailed overview. In this paper, we propose a rigorous method for analysing the behaviour of SPLs specified using the DOP approach. It is based on *Differential Symbolic Execution* (DSE) which facilitates identifying the semantic changes among products and allows fir describing their behavioural differences [3]. We use this analysis in order to generate test cases for software products, relative to the already-tested products and their test cases.

The remainder of this abstract is organised as follows: Section 2 outlines the problem of analysing SPLs and Section 3 presents our proposed approach for saving testing effort. An overview of preliminary results is given in Section 4. Finally, Section 5 outlines future work.

## 2 Problem Description

Verifying a product of an SPL involves the analysis of the core parts and a subset of the SPL's variability points. Checking another product of the same SPL may examine the same behaviour in the common core part and may cover a possibly overlapping subset of the variability points. Consequently, unnecessary redundant analysis steps are carried out, which results in unnecessarily excessive cost and time to market. Therefore, we would like to identify the featured behaviour of one product relative to another to minimise the SPL's analysis effort by avoiding retesting.

[1] See the product line hall of fame at http://splc.net/fame.html

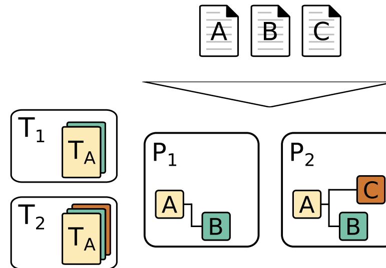



Figure 1: A Software Product Line including products $P_1$ and $P_2$, which are generated using delta modules $A$, $B$, and $C$, and verified by respective test suites $T_1$ and $T_2$.

To illustrate this problem, consider the delta-oriented SPL depicted in Figure 1. Its variability is specified in terms of delta modules $A$, $B$, and $C$, which generate software products $P_1$ and $P_2$. Within those products, the modules $A$ and $B$ are common to both products, whereas module $C$ is exclusive to product $P_2$. Syntactic changes, i.e., program code that is explicitly introduced by module $C$, can be easily identified using the DOP structure. Semantic changes, i.e., behaviour that is implicitly modified by the presence of module $C$, may be difficult to scrutinize though.

Consider the parameterised test suites $T_1$ and $T_2$ for the respective products $P_1$ and $P_2$, which cover the products' symbolic control-flow graphs. When exploiting the structure of the DOP approach, test suite $T_1$ covers all paths of the modules $A$ and $B$ within product $P_1$. Thus, verifying those modules' correctness in product $P_2$ becomes redundant. Therefore, verification in product $P_2$ should focus on the syntactic changes in module $C$. However, the structure of the DOP approach may be too imprecise to cover possible semantic changes in module $B$.

## 3 Proposed Approach

The proposed approach applies DSE to SPLs to identify the structural changes and to exploit those changes to select the corresponding test cases for re-execution. Ultimately, this allows for dispensing with excessive retesting of the inherited shared behaviour and provides opportunities for more effective verification trajectories of the exclusive behaviour.

Applying DSE to the SPL illustrated in Figure 1 results in the differential summary of product $P_2$. This information, denoted by $\delta_{P_2}$, describes the precise behavioural changes within the structure of $P_2$ when compared to the product $P_1$. Differential summary $\delta_{P_2}$ can then be used to identify a subset of parameterised unit test cases for the syntactical changes and to instantiate parameterised unit test cases for the semantical changes. Executing the subset of test cases covering the differential behaviour avoids redundant analysis and saves analysis effort.

Consider verifying products $P_1$ and $P_2$ of the SPL using the test suite $T_1$ and a subset of the test suite $T_2$. Then, the paths of the variability points $A$ and $B$ are traversed by $T_1$ (coloured in yellow and green, respectively). Similarly, the variability point $C$ comprising the syntactic changes to the behaviour of product $P_2$ is covered by test suite $T_2$ (coloured in red). By exploiting the differential summary, the retesting effort can focused on the semantically modified behaviour in module $B$.

In summary, the proposed approach allows for (a) selecting a subset of test cases by incorporating the identified differential behaviour, (b) optimising a given test suite by reducing its

number of test cases, and (c) guiding the SPL analysis by minimising its overall test effort. As a result, redundant analysis is bypassed leading to more a more efficient SPL analysis strategy.

## 4 Preliminary Results

We started to formalise an input language based on the semantics of the DOP approach DeltaJ [2]. The input language is transformed into the *Intermediate Verification Language* Boogie [1] that provides a symbolic execution engine. This modular process separates the concerns of encoding variability for an SPL and reasoning about the control-flow within its products and allows for applying the same technique to other SPL implementation frameworks. This also avoids over-fitting our approach to one specific SPL framework.

Based on the formalised input language, we defined the notion of *abstract* and *symbolic summaries* [3]. Abstract summaries over-approximate the behavior of a feature and the symbolic summary characterises the individual program execution paths of a given feature for all possible inputs.

We have currently outlined an algorithm for our formalised input language to create symbolic summaries for differential behaviour. In this manner, comparing the symbolic summaries for two software products allows for reasoning about their behaviour changes concerning their features. These changes can be used to modify the respective test cases both locally and also concerning those features that are accessed through the modified behavior.

## 5 Future Work

We are currently working on connecting the calculated summaries to a parameterised unit testing framework. We would like to put our preliminary results of our proposed approach together and evaluate (1) whether DSE is applicable to SPLs in order to efficiently derive the behavioural changes among different products, and (2) to what extent feature interactions including the added, modified, and deleted behavioural changes are covered by the automatically generated test cases.

## References

[1] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2006.

[2] Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. DeltaJ 1.5: Delta-oriented programming for Java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools*, pages 63–74. ACM, 2014.

[3] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237. ACM, 2008.

[4] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Software Product Lines: Going Beyond*, pages 77–91. Springer, 2010.

[5] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys (CSUR)*, 47(1):6, 2014.