REPORTS
IN
INFORMATICS

ISSN 0333-3590

24th Nordic Workshop on Programming Theory

NWPT 2012

Bergen Norway, 31 October - 2 November 2012

Abstracts

REPORT NO 403 **October 2012**

Department of Informatics

UNIVERSITY OF BERGEN

Bergen, Norway

This report has URL http://www.ii.uib.no/publikasjoner/texrap/pdf/2012-403.pdf

Reports in Informatics from Department of Informatics, University of Bergen, Norway, is available at http://www.ii.uib.no/publikasjoner/texrap/.

Requests for paper copies of this report can be sent to:

Department of Informatics, University of Bergen, Høyteknologisenteret,
P.O. Box 7800, N-5020 Bergen, Norway

Preface

This volume contains the abstracts[1] of the talks to be presented at the 24th Nordic Workshop on Programming Theory (NWPT 2012) to take place in Bergen, Norway, 31 October - 2 November 2012. The workshop is organized by the Department of Informatics, University of Bergen, and the Bergen University College.

The main goal of these workshops is to bring together researchers from Nordic and Baltic region (but not exclusively) active in the field of programming theory. Topics of the workshops cover traditional as well as emerging disciplines within programming theory. In particular, the workshops have encouraged young researchers wishing to enter the field and have often provoded valuable feedback for those already active for many years in this research area.

There will be 23 regular presentations at NWPT 2012. In addition the following three invited speakers will give a presentation: Anders Möller (Århus University, Denmark), Juan de Lara (Universidad Autonoma de Madrid, Spain) and Arne Styve (Offshore Simulator Centre, Ålesund, Norway).

Programme Committee

Luca Aceto, Reykjavík University, Iceland
Lars Birkedal, IT University of Copenhagen, Denmark
Einar Broch Johnsen, University of Oslo, Norway
Michael R. Hansen, Technical University of Denmark, Denmark
Yngve Lamo, Bergen University College, Norway
Kim G. Larsen, Aalborg University, Denmark
Bengt Nordström, Chalmers, University of Gothenburg, Sweden
Olaf Owe, University of Oslo, Norway
Paul Pettersson, Mälardalen University, Sweden
Gerardo Schneider, Chalmers, University of Gothenburg, Sweden
Andrei Sabelfeld, Chalmers, University of Gothenburg, Sweden
Tarmo Uustalu, Institute of Cybernetics, Estonia
Jüri Vain, Tallinn University of Technology, Estonai
Marina Waldén, Åbo Akademi University, Finland
Uwe Wolter, University of Bergen, Norway
Wang Yi, Uppsala University, Sweden

Local Organizing Committee

Uwe Wolter (co-chair) and Anya Helene Bagge, University of Bergen
Yngve Lamo (co-chair) and Lars Michael Kristensen, Bergen University College

We thank the authors of the submitted abstracts, the invited speakers and the members of the progamme committee for their help in making this workshop a successful scientific event.

Uwe Wolter and Yngve Lamo *Bergen, 31 October 2012*

[1] The program of NWPT 2012 is also included and the abstracts of the talks are ordered according to the program

NWPT 2012 – Programme – Wednesday 31. October	
08.30-09.15	**Registration**
09.15-09.30	**Opening and Welcome**
09.30-10.30	**Invited talk: Anders Møller** **Title: Static Analysis for JavaScript Web Applications**
10.30-11.00	*Coffee Break*
11.00-12.30	**Session 1: Program Analysis and Verification**
	Holger Siegel and Axel Simon: FESA: Fold- and Expand-based Shape Analysis
	Ka I Pun, Martin Steffen and Volker Stolz: Deadlock checking by race detection
	Julian Samborski Forlese and Cesar Sanchez: Bounded Model Checking for Regular Linear Temporal Logic
12.30-13.30	*Lunch*
13.30-15.00	**Session 2: Timed Systems**
	Étienne André and Shweta Garg: Robustness Analysis of Time Petri Nets
	Mingsong Lv, Nan Guan, Wang Yi and Ge Yu: Multicore Timing Analysis using Abstract Interpretation and Model Checking
	Mingsong Lv, Nan Guan, Wang Yi and Ge Yu: A Novel Cache PERSISTENCE Analysis for Worst-Case Execution Time Estimation
15.00-15.30	*Coffee Break*
15.30-17.00	**Session 3: Rewriting and Languages**
	Maja Tønnesen and Robert Glück: Semi Inversion of Conditional Constructor Term Rewriting Systems
	Denis Firsov and Tarmo Uustalu: Certified CYK parsing of context-free languages
	Eero Lassila. Tetrasystems: A Framework for String Generation Devices
19.00-21.00	**Reception by the Mayor - Schøtstuene**

NWPT 2012 – Programme – Thursday 1. November	
09.00-10.00	**Invited talk: Juan de Lara** **Title: Towards a more flexible Model-Driven Engineering**
10.00-10.30	Hamid Ebaditavallaei: Record Based Differential Privacy Budget
10.30-11.00	*Coffee Break*
11.00-12.30	**Session 4: Model-Driven Software Engineering and Synthesis**
	Florian Mantz and Gabriele Taentzer: Meta-model Evolution with Model Migration based on Graph Transformation
	Kent Inge Fagerland Simonsen, Lars Kristensen and Ekkart Kindler: Code Generation for Protocols from CPN models Annotated with Pragmatics
	Alexandre David, Kim Guldstrand Larsen and Zhengkui Zhang: Distributed Algorithms for Controller Synthesis
12.30-13.30	*Lunch*
13.30-14.30	**Invited talk: Arne Styve** **Title: Modern Software Development - A Synonym for "Re-inventing the wheel"**
14.30-15.00	Adrian Rutle and Hans Georg Schaathun: Model-Driven Engineering of Maritime Systems
15.00-15.30	*Coffee Break*
15.30-17.00	**Session 5: Semantics**
	Marco Patrignani and Dave Clarke: Fully Abstract Trace Semantics of Low-level Protection Mechanisms
	Cristian Prisacariu: Dynamic Structural Operational Semantics (preliminary report)
	Wusheng Wang, Gian Perrone and Thomas Hildebrandt: Petri Nets in Bigraphs Revisited
19.00-23.00	**Conference dinner at Fløyen folkerestaurant**

NWPT 2012 – Programme – Friday 2. November	
09.00-10.30	**Session 6: Reasoning and Analysis**
	Crystal Chang Din and Olaf Owe: Soundness of a Reasoning System for Asynchronous Communication with Futures
	Thi Mai Thuong Tran, Martin Steffen and Hoang Truong: Compositional Analysis of Resource Bounds for Software Transactions
	Fabrizio Biondi, Axel Legay, Bo Friis Nielsen and Andrzej Wasowski: Maximizing Entropy over Markov Processes
10.30-11.00	*Coffee Break*
11.00-12.30	**Session 7: Programming and Specification Languages**
	Magne Haveraaen: Literals in programming languages
	Elmo Todurov and Keiko Nakata: A finer module system for the ABS language
	Petter Sandvik: SPECTA - A Formal Specification Language for Content Transfer Algorithms
12.30-13.00	*Lunch*
13.00-14.00	PC meeting

Static Analysis for JavaScript Web Applications (Invited Talk)

Anders Møller

Aarhus University
amoeller@cs.au.dk

Abstract

JavaScript is the foundation of all modern browser-based web applications. As a programming language, JavaScript has roots in both object-oriented and functional programming. It contains many dynamic features that are not found in traditional mainstream languages. Most importantly, it has a flexible notion of objects: properties are added dynamically, the names of the properties are dynamically computed strings, the types of the properties are not fixed, and prototype relations between objects change during execution. Experimental studies have shown that these dynamic features in JavaScript are widely used.

The flexibility provided by these dynamic features has a price. For many kinds of programming errors that cause compilation errors or runtime errors in other languages, JavaScript programs keep on running, often with surprising consequences, which can make it difficult to locate the causes of the errors. It becomes challenging for the programmers to reason about the behavior of JavaScript programs without actually running them. To make matters worse, the language provides essentially no encapsulation mechanisms. Because of this language design, tool support is still limited and JavaScript programmers rely on tedious testing to a much greater extent than necessary with statically typed languages. Unlike the first scripts that appeared when JavaScript was introduced, modern JavaScript programs often contain thousands of lines of code, so it becomes increasingly important to develop better tool support for the JavaScript programmers.

This talk presents an overview of the TAJS project[1], which aims to build a static analysis tool for JavaScript. Our analysis tool infers an abstract state for each program point in a given JavaScript web application. Such an abstract state soundly models the possible states that may appear at runtime and can be used for detecting type-related errors and dead code. These errors often arise from wrong function parameters, misunderstandings of the runtime type coercion rules, or simple typos that can be tedious to find using testing. Our focus has first been on the abstract domain and dataflow constraints that are required for a sound and reasonably precise modeling of the basic operations of the JavaScript language itself and the native objects that are specified in the ECMAScript standard [3]. Next, to reason about web application code, we have built a model of the browser API, including the HTML DOM and the event system [2]. In parallel we have developed new techniques for interprocedural dataflow analysis to boost performance [4]. More recently, we have taken the first step of handling common patterns of code that is dynamically generated using the `eval` function [1].

[1] `http://www.brics.dk/TAJS/`

References

[1] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *Proc. 21st International Symposium on Software Testing and Analysis*, July 2012.

[2] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proc. 8th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2011.

[3] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, volume 5673 of *LNCS*. Springer, August 2009.

[4] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium*, volume 6337 of *LNCS*. Springer, September 2010.

FESA: Fold- and Expand-based Shape Analysis

Holger Siegel and Axel Simon

Technische Universität München, Institut für Informatik II, Garching, Germany
firstname.lastname@in.tum.de

Introduction. Proving the absence of `NULL`- and dangling pointer dereferences in heap-manipulating programs requires the ability to deal with an a priori unbounded number of heap-allocated memory cells. We present a static shape analysis that is able to prove the absence of `NULL`- and dangling pointer dereferences in standard algorithms on lists, trees and graphs. Heaps are modelled as a graph together with a numeric state that distinguishes the various configurations. The key insight is that summarizing heap cells reduces to summarization of the numeric state, leading to an analysis that lies at a sweet-spot between precision and simplicity.

Consider the C program in Fig. 1a) that constructs a singly linked list. We show how the allocated cells can be merged into a single summary node A, resulting in the heap in Fig. 1b). Before the loop in Fig. 1a) is entered for the first time, `x` and `y` both point to a single node allocated by `new_node()`. The corresponding heap in Fig. 2a) depicts program variables as diamonds and heap cells as circles. Figure 2b) shows the state after one iteration. Here, `y` points to the newly allocated heap cell B. Figure 2c) shows the state after another iteration.

Heap Representation. We represent sets of heaps using a single points-to map (a graph) and a numeric state which is merely a set of $\{0, 1\}$-vectors. The points-to map takes each heap cell or program variable to a set of points-to edges and is shown as a directed graph. Each points-to edge is then further qualified by a flag that is mapped to zero or to one by the numeric domain. In particular, a flag f_{AB} maps to one iff the edge from node A to node B exists. For instance, the heap before the loop consists of the points-to map shown in Fig. 2a) and the numeric state $\langle f_{xA}, f_{yA}\rangle \in \{\langle 1,1\rangle\}$. The heap after one iteration consists of the points-to map shown in Fig. 2b) and the numeric state $\langle f_{xA}, f_{yB}, f_{AB}\rangle \in \{\langle 1,1,1\rangle\}$. After another iteration, the heap consists of the points-to map in Fig. 2c) and the numeric state $\langle f_{xA}, f_{yC}, f_{AB}, f_{BC}\rangle \in \{\langle 1,1,1,1\rangle\}$.

These three heaps cannot be merged directly, since their sets of edges and nodes are different. We therefore make them *compatible* to each other by adding missing edges and nodes. Figure 3a) shows how adding nodes B and C and edges yB, yC, AB and BC to Fig. 2a) gives a compatible points-to graph with numeric state $\langle f_{xA}, f_{yA}, f_{yB}, f_{yC}, f_{AB}, f_{BC}\rangle \in \{\langle 1,1,0,0,0,0\rangle\}$. Figure 3b) and c) show how the heaps in Fig. 2b) and c) are made compatible with the corresponding numeric states $\{\langle 1,0,1,0,1,0\rangle\}$ and $\{\langle 1,0,0,1,1,1\rangle\}$, respectively. The merge of these compatible heaps is completed by joining the three numeric states into a single state $b := \{\langle 1,1,0,0,0,0\rangle\, \langle 1,0,1,0,1,0\rangle\, \langle 1,0,0,1,1,1\rangle\}$. The benefit of this representation is that the three heap configurations are all encoded by the graph in Fig. 3d) and the numeric state b.

Numeric Summaries. We summarize two nodes by calculating their common points-to information using the recently proposed *fold* and *expand* operations [4]. The *fold* operation is used when two heap cells are merged into a single summary cell. A symmetric *expand* operation duplicates these relations when a summary cell is materialized back into two heap cells, one of which is a concrete non-summary cell. The force of this approach is that both, *fold* and *expand*, retain relational information between points-to edges, thereby essentially inferring new shape invariants. For instance, given the vectors $\langle a,b,c,d,e\rangle \in \{\langle 0,0,0,1,1\rangle, \langle 1,0,1,1,0\rangle\}$. Folding dimensions

a)
```
x = y = new_node();
while(rnd()) {
  *y = new_node();
  y = *y;
}
```

b)

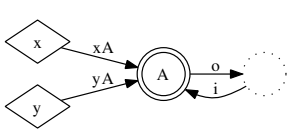
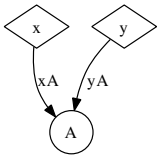
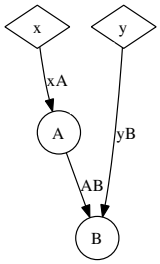


Figure 1: allocating a linked list

a)



b)



c)

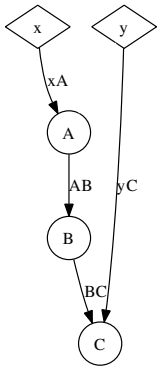


Figure 2: calculating the state in the `while`-loop

d, e onto b, c gives $\langle a, b, c\rangle \in \{\langle 0,0,0\rangle, \langle 0,1,1\rangle, \langle 1,0,1\rangle, \langle 1,1,0\rangle\}$, retaining that $b = c$ iff $a = 0$. Re-expanding d, e from b, c gives $\{\langle 0, b, c, d, e\rangle \mid b = c \wedge d = e\} \cup \{\langle 1, b, c, d, e\rangle \mid b \neq c \wedge d \neq e\}$.

Creating Summaries. We summarize the graph in Fig. 3d) and state b into a graph where all list nodes are summarized into one node as shown in Fig. 1b). To this end, our analysis uses the points-to graph to determine which edges to overlay and then applies the relational *fold* operation several times in order to summarize the corresponding dimensions. As a result, the dimensions $\langle f_{xA}, f_{yA}, f_{yB}, f_{yC}, f_{AB}, f_{BC}\rangle$ corresponding to Fig. 3d) are mapped to the dimensions $\langle f_{xA}, f_{yA}, f_i, f_o\rangle$ corresponding to Fig. 1b). This summarized heap represents the common points-to properties of all nodes, where the edges f_i and f_o represent the in- and outgoing edges connecting A with another instance of A. This instance is drawn with a dotted circle and is called the *ghost node* of A. Interestingly, if we were to summarize a list with up to four nodes, we would obtain the same summarized heap, that is, the summarized heap is already a fixpoint for the loop. Indeed, this summary represents a singly linked list of arbitrary size. The analysis has thus inferred that the loop constructs a singly linked list that commences in `x` and ends in the node pointed to by `y`. The numeric state of the fixpoint is quite subtle and describes the various rôles the summary node can take on: the list head A in Fig. 3a); the list head A in b), c); the unreachable nodes B, C in a) and C in b); the final node B in b) and C in c); and the inner node B in c). The key observation is that the relational *fold* is able to automatically synthesize these rôles and that the Boolean function maintains the distinction between them, thereby inferring very strong shape invariants.

Operations on Summaries. We now consider the loop in Fig. 4a) that deallocates a linked list. In our analysis, a summary node is materialized each time it is accessed. Thus, all modifications to heap nodes are performed on materialized nodes, which ensures that the abstract transformers are easy to derive since they closely follow the concrete transformers. Consider the assignment `x = *x` in the third line. Since the dereference `*x` would access a summary node, our analysis materializes a node D from the summary before executing the assignment. This materialization results in a heap with the points-to set depicted in Fig. 4b). In this particular case it is possible to free the node D by simply removing it from the graph. After `z` goes out of scope, the resulting graph is already compatible with that at the loop head. A fixpoint for

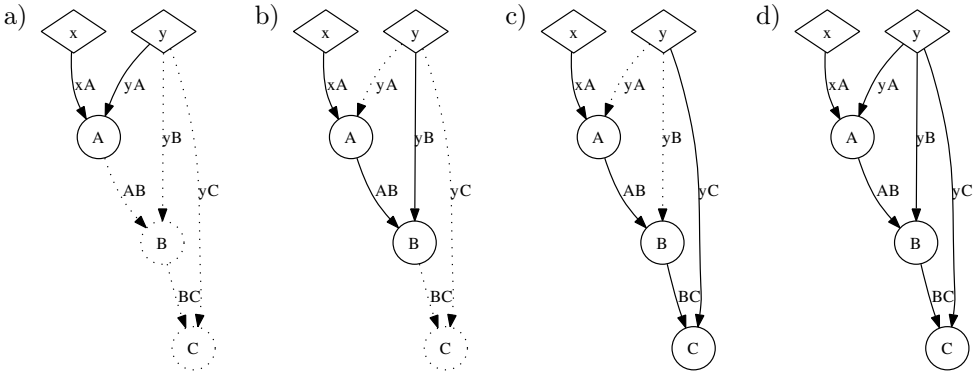


Figure 3: joining states

a)
```
do {
    z = x;
    x = *x;
    free(z);
} while(x);
```

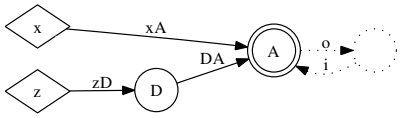
b)


Figure 4: deallocating a linked list

the numeric domain is observed after one further iteration. When evaluating the expression `*x`, the analysis checks that the numeric domain maps at least one edge in the points-to set of `x` to one. If this is not the case, a warning is emitted, stating that `x` can be `NULL`. In the example, our analysis is precise enough to verify the absence of `NULL`-pointer dereferences. Indeed, it can infer invariants that distinguish lists from trees from graphs when additionally tracking simple reference counters. In contrast to other analyses [2, 3], no extra effort is needed to make our analysis robust with respect to variations of these basic data structures (position of pointer fields, use of sentinel nodes instead of `NULL` values or the use of back pointers).

Implementation A numeric state $b \subseteq \{0, 1\}^n$ can be represented by a Boolean formula over n variables. We therefore represent heaps as a tuple consisting of a points-to graph and a single formula φ for b in conjunctive normal form. Unused variables in the numeric state are removed using the projection method of Brauer et al. [1], which is based on SAT-solving. By summarizing heap cells on loop entry, we are able to verify common list and tree algorithms.

References

[1] J. Brauer, A. King, and J. Kriener. Existential Quantification as Incremental SAT. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, LNCS, page 16. Springer, July 2011.

[2] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by means of Bi-Abduction. In *Principles of Programming Languages*, Savannah, Georgia, USA, January 2009. ACM.

[3] B. McCloskey, T. Reps, and M. Sagiv. Statically Inferring Complex Heap, Array, and Numeric Invariants. In *Static Analysis Symposium*, volume 6337 of *LNCS*, pages 71–99, Perpignan, France, 2010. Springer.

[4] H. Siegel and A. Simon. Summarized Dimensions Revisited. In L. Mauborgne, editor, *Workshop on Numeric and Symbolic Abstract Domains*, ENTCS, Venice, Italy, September 2011. Springer.

Deadlock checking by data race detection

Ka I Pun[1], Martin Steffen[1] and Volker Stolz[1,2]

[1] Department of Informatics, University of Oslo, Norway
[2] United Nations University—Intl. Inst. for Software Technology, Macao

Motivation

In concurrent programs, locks are commonly used to avoid simultaneous access to shared resources. A deadlock occurs when multiple processes wait for locks in a cycle. The competition between these processes for the access to shared resources can be interpreted as the *race* of the last two processes to close the deadlock cycle.

Instead of developing/using a custom deadlock checker, we can thus use existing race checkers (e.g. `Chord` [2] and `Goblint` [5]) for *static* race detection after instrumenting a program with suitable shared variable accesses.

To facilitate the instrumentation, we present a type system which captures so-called *second lock points*, a static over-approximation of program points where deadlocks can actually manifest themselves. This type-information is used to transform the program into its corresponding instrumented version with conflicting accesses to race variables. We show the soundness of our approach, which is that for a program with a deadlock, a race analysis will report a race on the transformed program, and that the transformation preserves deadlocks.

Type System and Transformation

Our type system algorithmically tracks which locks are actually handled in the interactions by annotating each lock with the corresponding program point π of its creation. Furthermore, it tracks the relative change to the lock count, denoted as $\Delta \to \Delta'$, through each statement. The type system uses constraints [1, 3] to derive the *smallest* possible type (in terms of originating locations) for each variable of lock-type in the program.

The instrumentation transforms the program by comparing the static analysis information with the desired cycle. The cycle $\Delta_c$is expressed in terms of a "ring" of (abstract) processes, each one *holding* a particular lock and *requesting* another one. This corresponds to a deadlock/deadlocked configuration [4] with a heap σ and processes P where every involved process p has $waits(\sigma \vdash P, p, l)$ is blocked on taking l. (Such a deadlock can occur in various places in a program.)

We call a program location a static second lock point (SLP) for Δ_c, where the analysis information *from the type* indicates that both the lock "we" are trying to take is involved in the cycle, and we already hold the corresponding other lock in Δ. As due to the over-approximation of locations a lock-statement can refer to a set of possible lock locations, instrumentation must occur if *any* of the potential locks is involved in the cycle.

The judgement of the type system is given as

$$\Gamma \vdash e : T :: \Delta \to \Delta'; C$$

which means that the expression e has type T and has the relative change to lock counts from Δ to Δ' under the constraint C.

Below, we show some of the relevant typing rules of our system for a functional language with locks. Lock creation in rule T-NEWL introduces an abstract location which is henceforth tracked in the constraints.

Spawning a thread in rule T-SPAWN has no effect on the caller, and the premise of the rule checks well-typedness of the expression being spawned. Note that for that expression, since it will be executed in a new thread, all locks are assumed to be initially free (indicated by $\bullet$).

Rules T-LOCK and T-UNLOCK describe the operations of locking and unlocking, simply counting up, resp. down the lock counter, setting the post-condition to $\Delta \oplus \rho$, resp. $\Delta \ominus \rho$ where $\oplus$ and $\ominus$ record the change for the particular abstract lock. Note that the constraints can be solved *after* type checking and *before* calculating the Δs.

$$\frac{\rho \textit{ fresh}}{\Gamma \vdash \texttt{new}_\pi\ \mathrm{L} : \mathrm{L}^\rho :: \Delta \to \Delta; \rho \supseteq \{\pi\}}\ \text{T-NEWL} \qquad \frac{\Gamma \vdash e : \hat{T} :: \bullet \to \Delta_2; C}{\Gamma \vdash \texttt{spawn}\ e : \texttt{Thread} :: \Delta_1 \to \Delta_1; C}\ \text{T-SPAWN}$$

$$\frac{\Gamma \vdash v : \mathrm{L}^\rho :: \Delta_1 \to \Delta_1; \emptyset \qquad \Delta_2 = \Delta_1 \oplus \rho}{\Gamma \vdash v.\ \texttt{lock} : \mathrm{L}^\rho :: \Delta_1 \to \Delta_2; \emptyset}\ \text{T-LOCK} \qquad \frac{\Gamma \vdash v : \mathrm{L}^\rho :: \Delta_1 \to \Delta_1; \emptyset \qquad \Delta_2 = \Delta_1 \ominus \rho}{\Gamma \vdash v.\ \texttt{unlock} : \mathrm{L}^\rho :: \Delta_1 \to \Delta_2; \emptyset}\ \text{T-UNLOCK}$$

Whether or not to prepend a race-variable to a single `lock`-statement depends on if it is a second lock point or not.

The number of possible deadlocks does not depend on the number of abstract lock locations occurring in the program: even a single location, i.e. all locks stem from the same `new`-statement, is sufficient to form deadlocks of arbitrary length.

The number of introduced race variables increases (by one) for each cycle that we would like to check for. It is possible to instrument a program for a set of potential cycles without the different race variables interfering with each other. Alternatively, for scalability it is possible to e.g. parallelize race checker-runs on programs instrumented for different (sub-sets of) cycles.

We illustrate our approach using the dining philosophers:

```
let l_1 = new^{π_1} L; ...; l_n = new^{π_n} L // create all locks
    phil = fun F(x,y) . ( x.lock; y.lock;
                          /* think */
                          y.unlock; x.unlock; F(x,y) )
in spawn(phil(l_1,l_2)); ... ; spawn(phil(l_n,l_1))
```

As *type* for the philosopher-function, we obtain that each of the two arguments of lock-type has type $\mathrm{L}^{\{\pi_1,\ldots,\pi_n\}}$, i.e., they may origin from *any* of the `new`-statements. As the function is *balanced*, there is no relative change, hence the pre-Δ corresponding to the *first* `lock`-statement will always be empty. Thus this statement is correctly never identified as a sSLP. The second `lock`-statement however will be instrumented by the transformation, and thus able to trigger a race if accessed simultaneously.

For the *fixed* dining philosophers, where e.g. the last philosopher accesses the locks in order l_1, l_n, although *the types change* for both locks (the first lock-set no longer contains π_n, the second no longer π_1), we still obtain the same instrumentation, and hence the same report about races, even though the program does not have any concrete deadlock.

Gate locks To increase precision (as a race can be already triggered by just *two* processes), we add gate-locks at appropriate places before the shared variable-accesses. This also requires reducing the amount of code shared between processes, as the transformation needs to be able to create *distinct* instrumentations for correlated SLPs. In the above philosophers example, all processes share the same function definition, so the `let` has to be pushed into the `spawn`s. This is here already sufficient to

disentangle the confusion about the origin of the locks in the types. In general though, the introduced gate locks are necessary to prevent the detection of partial cycles, but leave the overall behaviour with regard to actual deadlocks unchanged – they only influence how race variables are accessed.

Results

We give a formal description for our type system and prove the soundness of our approach, i.e., a program with a (potential) deadlock will be reported as having a race in its version instrumented/transformed for that particular cycle. Deadlocks can only occur at static second lock points for each of the involved processes. The transformation guarantees that each of these SLPs is protected by the same race variable for that cycle which implies a race between any two of the involved processes.

Acknowledgements Supported by the ARV grant of the Macao Science and Technology Development Fund.

References

[1] C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.

[2] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, pages 308–319. ACM, 2006.

[3] F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[4] K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. *Journal of Logic and Algebraic Programming*, 81(3):331–354, Mar. 2012.

[5] H. Seidl and V. Vojdani. Region analysis for race detection. In J. Palsberg and Z. Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 171–187. Springer-Verlag, 2009.

Bounded Model Checking for Regular Linear Temporal Logic

Julián Samborski-Forlese[1] and César Sánchez[1,2]

[1]IMDEA Software Institute, Madrid, Spain [2]Institute for Applied Physics (CSIC), Spain

Abstract

Bounded model checking (BMC) is an effective method to perform temporal formal verification using LTL as a specification. However, LTL has limited expressivity and some important properties cannot be described in this logic.

Regular Linear Temporal Logic (RLTL) is an extension of LTL equipped with constructs that allow to express ω-regular expressions, thus extending the expressive power of LTL to all ω-regular languages. In this paper we present a BMC algorithm for RLTL, carried out as a two stage process (for the finite and temporal parts of RLTL separately), and prove the correctness of this method.

1 Introduction

Bounded model checking for LTL [2] is a symbolic model checking technique based on SAT. The basic idea of BMC is to restrict the counterexample traces to contain up-to a k different states, and produce a propositional boolean formula that is satisfiable iff such a counterexample exists. The parameter k is then incremented until either a counterexample is found or all the state space has been explored.

In this paper, we present a novel BMC algorithm for the logic RLTL, a logic that properly extends LTL and ω-regular expressions. First we define a bounded semantics for RLTL, and then provide a translation from RLTL expressions on ultimately periodic words into propositional formulas.

RLTL in a Nutshell Regular Linear Temporal Logic [6, 7] is a logic for specifying ω-regular languages. The availability of both regular expressions and LTL-like operators and the absence of fix-point binders or automata constructors makes RLTL an intuitive extension of LTL. The logic RLTL is defined in two stages, (like PSL [5] or ForSpec [1]). The first stage consists on a variation of regular expressions that define finite non-empty *segments* of infinite words. Starting from atomic propositions p the following are regular expressions:

$$\alpha ::= \alpha + \alpha \quad | \quad \alpha \,;\, \alpha \quad | \quad \alpha * \alpha \quad | \quad p$$

The operators $+$, ; and $*$ are the standard union, concatenation and binary Kleene-star. The second stage uses regular expressions to build RLTL expressions φ:

$$\varphi ::= \varnothing \quad | \quad \varphi \vee \varphi \quad | \quad \neg\varphi \quad | \quad \alpha \,;\, \varphi \quad | \quad \varphi|\alpha\rangle\!\rangle\varphi \quad | \quad \varphi|\alpha\rangle\varphi$$

Semantics Given an ω-word w and $i, j \in \mathbb{N}$, we use $w[i]$ for the symbol at *position* i in w, and (w, i, j) for the *segment* of w between positions i and j. A *pointed word* is a pair (w, i) formed by a word w and a position i. The formal semantics of regular expressions is defined as a binary relation $\vDash_{RE}$ between segments and regular expressions. For example, the semantics of $r * s$

establish that $(w,i,j) \vDash_{RE} r * s$ whenever (a) either $(w,i,j) \vDash_{RE} s$, or (b) for some sequence $(i_0 = i, i_1, \dots i_m)$ and for all $k \in \{0,..,m-1\}$, $(w,i_k,i_{k+1}) \vDash_{RE} r$ and $(w,i_m,j) \vDash_{RE} s$.

The semantics of RLTL relate expressions and pointed words. For example, for the power operator: $(w,i) \vDash x|r\rangle\!\rangle y$ whenever (a) either $(w,i) \vDash y$; or (b) for some sequence $(i_0 = i, i_1, \dots i_m)$, $(w,i_k,i_{k+1}) \vDash_{RE} r$ and $(w,i_k) \vDash x$, and $(w,i_m) \vDash y$.

Bounded Semantics For $l \leq k$, we call $w \in \Sigma^\omega$ a (k,l)-looping word if $w = uv^\omega$ with $u = w_0 w_1 \dots w_{l-1}$ and $v = w_l w_{l+1} \dots w_{k-1}$. We simply say that w is a k-looping word if there exists $l \in \mathbb{N}$ with $l \leq k$ such that w is a (k,l)-looping word. The *bounded semantics* for RLTL is an approximation of the unbounded semantics. Given a bound k and a word w, if w is a k-looping word, we apply the unbounded semantics. Otherwise, we use the bounded semantics, which only considers the segment $(w,0,k)$.

Definition 1 (Bounded Semantics for Looping Words)**.** *Let $w \in \Sigma^\omega$ be a k-looping word and x an RLTL expression. We say that x holds in w with bound k (i.e., $w \vDash_k x$) whenever $w \vDash x$.*

Definition 2 (Bounded Semantics for Non-Looping Words)**.** *Let $i < k$, and let $w \in \Sigma^\omega$ be a non-k-looping word. An RLTL expression x holds in w with bound k (i.e., $(w,i) \vDash_k x$):*

$(w,i) \vDash_k x \vee y$	*whenever either* $(w,i) \vDash_k x$ *or* $(w,i) \vDash_k y$
$(w,i) \vDash_k r; y$	*whenever for some position* $i \leq j < k$, $(w,i,j) \vDash_{RE} r$ *and* $(w,j) \vDash_k y$
$(w,i) \vDash_k x|r\rangle\!\rangle y$	*whenever* $(w,i) \vDash_k y$ *or for some sequence* $(i_0 = i, i_1, \dots i_m < k)$ $(w,i_j,i_{j+1}) \vDash_{RE} r$ *and* $(w,i_j) \vDash_k x$, *and* $(w,i_m) \vDash_k y$
$(w,i) \vDash_k x|r\rangle y$	*whenever* $(w,i) \vDash_k x|r\rangle\!\rangle y$ *holds*

Uncertainty can arise in negation and in the infinite behavior of the weak power operator $(x|r\rangle y)$. These cases are approximated by **false**. The key result is the following soundness lemma.

Lemma 1. *Let φ be an RLTL expression and $w \in \Sigma^\omega$, if $w \vDash_k \varphi$ then $w \vDash \varphi$.*

2 Encoding into SAT

We reduce the bounded model checking problem to propositional satisfiability in order to be able to use efficient SAT solvers [4]. We provide a translation from RLTL expressions into SAT formulas for looping words. For this translation we need two operators based on the notion of *derivatives* [3] of regular expressions. Given a word $w \in \Sigma^*$ and a set X of regular expressions, $\mathsf{deriv}_w(X)$ is the set of derivatives of every element of X with respect to w. We also use $\mathsf{DL}^*(w,r)$ for the derivatives of r with respect to w^*.

A Translation for Regular Expressions The following recursive definition allows to compute a propositional formula for a given segment and a regular expression, by simply unrolling all the possible cases. Let k,i,j be such that $i < j \leq k$, let $w \in \Sigma^\omega$ be a word and r a basic regular expression r. Then:

$$
\begin{aligned}
{}^i[\![\lambda]\!]_k^j &= j = i & {}^i[\![p]\!]_k^j &= p(w_i) \wedge j = i+1 \\
{}^i[\![r+s]\!]_k^j &= {}^i[\![r]\!]_k^j \vee {}^i[\![s]\!]_k^j & {}^i[\![r\,;s]\!]_k^j &= \bigvee_{n=i+1}^{j-1} \left({}^i[\![r]\!]_k^n \wedge {}^n[\![s]\!]_k^j \right) \\
{}^i[\![r*s]\!]_k^j &= {}^i[\![s]\!]_k^j \vee \bigvee_{n=i+1}^{j-1} \left({}^i[\![r]\!]_k^n \wedge {}^n[\![r*s]\!]_k^j \right)
\end{aligned}
$$

The following lemma establishes the soundness of this translation with respect to the semantics of basic regular expressions over segments of infinite words.

Lemma 2 (Soundness). *Let $w \in \Sigma^\omega$ be a word, $i < j$ be two positions and r be a basic regular expression, and s be the derivative of r with respect to (w, i, j). Then $\lambda \in \mathcal{L}(s)$ iff ${}^{i}[\![r]\!]^{j}_{k}$.*

We now generalize the translation of basic regular expressions to deal with looping ("${}^{i}_{l}[\![\cdot]\!]^{j}_{k}$"). Given an infinite word $w = w_{0l}\,(w_{lk})^\omega$, we establish precisely when the regular expression matches, respectively, the segment w_{ij} or the set of segments $w_{ik}\,(w_{lk})^*\,w_{lj}$ depending on whether or not $i < j$.

Definition 3 (Translation of a Regular Expression for a Loop). *Let $l < i, j \leq k$ with $i \neq k$ let $w \in \Sigma^\omega$ be a (k, l)-looping word, and let r be a regular expression*

$$ {}^{i}_{l}[\![r]\!]^{j}_{k} = {}^{i}[\![r]\!]^{j}_{k} \vee \bigvee_{s \in \mathsf{DL}^*(\partial_{w_{ik}}(r))} {}^{l}[\![s]\!]^{j}_{k}. $$

Lemma 3 (Soundness). *Let $l \leq i, j \leq k$ and $i \neq k$, let $w \in \Sigma^\omega$ be a (k, l)-looping word, and let r be a regular expression. Then, $\mathcal{L}(w_{ij} + w_{ik}\,;\, w_{lk} * w_{lj}) \cap \mathcal{L}(r) \neq \varnothing$ if and only if ${}^{i}_{l}[\![r]\!]^{j}_{k}$.*

A Translation for RLTL Formulas The translation of RLTL into propositions uses the previous translations for basic regular expressions. Given a (k, l)-looping word, if a position i is such that $l \leq i \leq k$, we say that i is inside the loop. Otherwise, we say that i is outside the loop. Depending on whether or not the starting position of the segment being translated is inside the loop, we have two different translations of a temporal formula x.

Definition 4 (Inside a Loop). *Let $l \leq i < k$, let w be a (k, l)-looping word, and let x be an RLTL expression:*

$$ {}^{i}_{l}[\![\varnothing]\!]_k = \mathbf{false} \qquad {}^{i}_{l}[\![\neg x]\!]_k = \neg {}^{i}_{l}[\![x]\!]_k \qquad {}^{i}_{l}[\![x \vee y]\!]_k = {}^{i}_{l}[\![x]\!]_k \vee {}^{i}_{l}[\![y]\!]_k $$

$$ {}^{i}_{l}[\![r\,;x]\!]_k = \left(\bigvee_{j=i+1}^{k-1} {}^{i}[\![r]\!]^{j}_{k} \wedge {}^{j}_{l}[\![x]\!]_k\right) \vee \left(\bigvee_{j=l}^{k-1} {}^{l}_{l}[\![\partial_{w_{ik}}(r)]\!]^{j}_{k} \wedge {}^{j}_{l}[\![x]\!]_k\right) $$

$${}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_k = \mathsf{F}^{k-l}\left({}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_k\right), \text{ where } \mathsf{F}^0\left({}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_k\right) = \mathbf{false}, \text{ and}$$

$$ \mathsf{F}^M\left({}^{i}_{l}[\![x|r\rangle\!\rangle y]\!]_k\right) = {}^{i}_{l}[\![y]\!]_k \vee {}^{i}_{l}[\![x]\!]_k \wedge \bigvee_{j=l+1}^{k-1} \left({}^{i}_{l}[\![r]\!]^{j}_{k} \wedge \mathsf{F}^{M-1}\left({}^{j}_{l}[\![x|r\rangle\!\rangle y]\!]_k\right)\right) $$

$${}^{i}_{l}[\![x|r\rangle y]\!]_k = \mathsf{G}^{k-l}\left({}^{i}_{l}[\![x|r\rangle y]\!]_k\right), \text{ where } \mathsf{G}^0\left({}^{i}_{l}[\![x|r\rangle y]\!]_k\right) \mathbf{true}, \text{ and}$$

$$ \mathsf{G}^M\left({}^{i}_{l}[\![x|r\rangle y]\!]_k\right) = {}^{i}_{l}[\![y]\!]_k \vee {}^{i}_{l}[\![x]\!]_k \wedge \bigvee_{j=l+1}^{k-1} \left({}^{i}_{l}[\![r]\!]^{j}_{k} \wedge \mathsf{G}^{M-1}\left({}^{j}_{l}[\![x|r\rangle y]\!]_k\right)\right) $$

Definition 5 (Outside a Loop). *Let $i \leq l < k$, let w be a (k, l)-looping word, and let x be an RLTL expression,*

$$ {}^{i}_{l}\langle\!\langle\varnothing\rangle\!\rangle_k = \mathbf{false} \qquad {}^{i}_{l}\langle\!\langle\neg x\rangle\!\rangle_k = \neg {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_k \qquad {}^{i}_{l}\langle\!\langle x \vee y\rangle\!\rangle_k = {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_k \vee {}^{i}_{l}\langle\!\langle y\rangle\!\rangle_k $$

$$ {}^{i}_{l}\langle\!\langle r\,;x\rangle\!\rangle_k = {}^{l}_{l}[\![\partial_{w_{il}}(r)\,;x]\!]_k \vee \bigvee_{j=i+1}^{l-1} \left({}^{i}[\![r]\!]^{j}_{k} \wedge {}^{j}_{l}\langle\!\langle x\rangle\!\rangle_k\right) $$

$$ {}^{i}_{l}\langle\!\langle x|r\rangle\!\rangle y\rangle\!\rangle_k = {}^{i}_{l}\langle\!\langle y\rangle\!\rangle_k \vee \bigvee_{j=i+1}^{l-1} \left({}^{i}[\![r]\!]^{j}_{k} \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_k \wedge {}^{j}_{l}\langle\!\langle x|r\rangle\!\rangle y\rangle\!\rangle_k\right) \vee \left({}^{l}_{l}[\![x|\partial_{w_{il}}(r)\rangle\!\rangle y]\!]_k \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_k\right) $$

$$ {}^{i}_{l}\langle\!\langle x|r\rangle y\rangle\!\rangle_k = {}^{i}_{l}\langle\!\langle y\rangle\!\rangle_k \vee \bigvee_{j=i+1}^{l-1} \left({}^{i}[\![r]\!]^{j}_{k} \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_k \wedge {}^{j}_{l}\langle\!\langle x|r\rangle y\rangle\!\rangle_k\right) \vee \left({}^{l}_{l}[\![x|\partial_{w_{il}}(r)\rangle y]\!]_k \wedge {}^{i}_{l}\langle\!\langle x\rangle\!\rangle_k\right) $$

The main algorithm starts at the beginning of the word by applying the translation outside the loop and constructs the SAT formula bottom-up. Formally, for a value k, an RLTL expression x and a k-looping word w the translation $[\![w, x]\!]_k$ is ${}^{0}_{l}\langle\!\langle x\rangle\!\rangle_k$. The soundness of this translation is stated by the following theorem.

Theorem 1 (Soundness). *The formula $[\![w, x]\!]_k$ is satisfiable if and only if $w \vDash_k x$.*

References

[1] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2):140–164, 2003.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'99)*, LNCS, pages 193–207. Springer, 1999.

[3] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, 1964.

[4] M. Davis and H. Putnam. A computing proc. for quantification theory. *JACM*, 7:201–215, 1960.

[5] D. Fisman, C. Eisner, and J. Havlicek. *Formal syntax and Semantics of PSL: Appendix B of Accellera Property Language Reference Manual, Version 1.1*, March 2004.

[6] M. Leucker and C. Sánchez. Regular linear temporal logic. In *Proc. of ICTAC'07*, volume 4711 of *LNCS*, pages 291–305. Springer, 2007.

[7] C. Sánchez and M. Leucker. Regular linear temporal logic with past. In *Proc. of VMCAI'10*, volume 5944 of *LNCS*, pages 295–311. Springer, 2010.

Robustness Analysis of Time Petri Nets

Étienne André[1] and Shweta Garg[2]

[1] Université Paris 13, Sorbonne Paris Cité, LIPN, France
Etienne.Andre@lipn.univ-paris13.fr
[2] Dept. of Computer Science, IIT Bombay, Mumbai, India
ShwetaGarg@cse.iitb.ac.in

Abstract

Given a parametric time Petri net with inhibitor arcs and a valuation of the timing requirements seen as parameters, we propose a method synthesizing a constraint on these parameters guaranteeing the same set of traces as for the reference valuation. This gives a quantitative measure of the robustness of the system for linear time properties.

1 Introduction

Real-time concurrent systems are often characterized with a set of timing requirements. Even when the correctness for a given set of such requirements has been proved (e.g., using Uppaal [LPY97]), the corresponding implementation may turn incorrect because the requirements in practice may be slightly different. For example, a system where some component performs an action for, say, 2 seconds can be implemented with a time greater but very close to 2 (say, 2.0001 s), in which case the formal guarantee may not hold anymore.

Many approaches in the literature (see [Mar11] for a survey) consider that the clock (or time) is varying, and measure the robustness of the system w.r.t. this varying clock. Here, we consider that the timing requirements of the system may be subject to some drift in practice. We use the formalism of PITPNs (parametric time Petri nets with inhibitor arcs, see, e.g., [TLR09]): this formalism allows one to reason in a *parametric* manner, by considering that the requirements are parameters (unknown constants). We extend to PITPNs the *inverse method* (initially defined in the setting of parametric timed automata [ACEF09]); this algorithm synthesizes a set of constraints on the parameters, which gives a quantitative measure of the robustness of a time Petri net with inhibitor arcs.

2 Parametric Time Petri Nets with Inhibitor Arcs

We mostly use here the definition of [TLR09], where a PITPN $\mathcal{P}$ is a standard Petri net extended with firing times of the form $[a, b]$ where a, b are taken from a set of *temporal parameters* P, that are unknown constant taking their values among rationals. A firing time $[a, b]$ means that the transition must be fired at least a and at most b units of time after it is enabled, i.e., after enough tokens are available in the input places. PITPNs also feature inhibitor arcs, that disable a transition when a token is present. This notion is somehow similar to *stopwatches*, since it blocks the elapsing of time, from the transition point of view. K_{init} is a constraint on P giving the initial *domain of the parameters*, and must at least specify that the minimum bounds of the firing intervals are

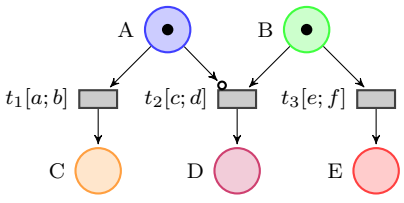


Figure 1: A PITPN

inferior to the maximum bounds. Figure 1 shows an example of PITPN, where the firing times of t_1, t_2 and t_3 are $[a, b]$, $[c, d]$, and $[e, f]$, respectively. Transition t_2 is inhibited by A.

Given a set X of local firing times of transitions and a set P of parameters, a constraint D over X and P is a conjunction of linear inequalities on X and P. We denote by $D{\downarrow_P}$ the constraint over P obtained from D after elimination of the firing times.

The reachable states of a PITPN are *parametric state-classes* $\mathbf{c}$, i.e., pairs (M, D) where M is a marking of the net and D is a firing domain, that is, a constraint over X and P. We consider a (classical) semantics where transitions *must* fire if they can; for example, in Figure 1, t_1 must fire before t_3 if $b < e$, t_3 must fire before t_1 if $f < a$, and both orders are possible otherwise. Given a valuation π, a class (M, D) is said to be π-*compatible* if $\pi \models D{\downarrow_P}$, and π-incompatible otherwise.

Given a run r of $\mathcal{P}$ of the form $(M_0, D_0) \overset{a_0}{\Rightarrow} \cdots \overset{a_{n-1}}{\Rightarrow} (M_n, D_n)$, the *trace associated with* r is the alternating sequence of markings and actions $M_0 \overset{a_0}{\Rightarrow} \cdots \overset{a_{n-1}}{\Rightarrow} M_n$. The *trace set* of $\mathcal{P}$ is the set of all traces associated with the runs of $\mathcal{P}$. $Post_{\mathcal{P}(K)}(C)$ (resp. $Post^i_{\mathcal{P}(K)}(C)$) is the set of classes reachable from a set C of classes in exactly one step (resp. i steps) under constraint K. Given a PITPN $\mathcal{P}$ and a valution π, we denote by $\mathcal{P}[\pi]$ the ITPN where each occurrence of a parameter has been replaced by its constant value as in π.

3 The Inverse Method for Time Petri Nets

3.1 Principle

We introduce in Algorithm 1 the inverse method *IM* for PITPNs. Given a PITPN $\mathcal{P}$ and a reference parameter valuation π_0, *IM* synthesizes a constraint K_0 on P such that, for all $\pi \models K_0$, $\mathcal{P}[\pi_0]$ and $\mathcal{P}[\pi]$ have the same trace sets. Starting from the initial class $\mathbf{c}_0$, *IM* iteratively computes state classes. When a π_0-incompatible class is found, an incompatible inequality is nondeterministically selected within the projection of the constraint onto P (line 5); its negation is then added to K (line 6). The set of reachable states is then updated. When all successor classes have been reached before (line 8), *IM* returns the intersection of the projection onto the parameters of the constraints associated with all the reachable classes.

Algorithm 1: $IM(\mathcal{P}, \pi_0)$

```
1  i ← 0 ;  K ← K_init ;  C ← {c_0}
2  while true do
3      while ∃ π0-incompat. classes in C do
4          Select a π0-incompatible class
           (M, D) of C
5          Select a π0-incompatible J in D↓P
6          K ← K ∧ ¬J
7          C ← ⋃_{j=0}^{i} Post^j_{P(K)}({c_0})
8      if Post_{P(K)}(C) ⊆ C then
9          return K0 ← ⋂_{(M,D)∈C} D↓P
10     i ← i + 1 ;  C ← C ∪ Post_{P(K)}(C)
```

3.2 Application to a Simple Example

Let us apply *IM* to the PITPN of Figure 1 with a π_0 where a, b, c, d, e, f take as values $5, 6, 3, 4, 1, 2$ respectively. We have $K_{init} = a \leq b \wedge c \leq d \wedge e \leq f$. For sake of conciseness, we always project D onto P (hence we omit the firing times in X). The initial class is $\mathbf{c}_0 = (\text{AB}, K_{init})$, where AB denotes that places A and B both contain a token. From $\mathbf{c}_0$, one can reach $\mathbf{c}_1 = (\text{AE}, K_{init} \wedge b \geq e)$, and a class $(\text{CB}, K_{init} \wedge f \geq a)$. The former is π_0-compatible, but not the latter because $f \geq a$ is not true in π_0 ($2 \not\geq 5$). Hence, we add $f < a$ to K, and go on. From $\{\mathbf{c}_0, \mathbf{c}_1\}$, one can reach $\mathbf{c}_2 = (\text{CE}, K_{init} \wedge b \geq e \wedge f < a)$, which is π_0-compatible. Then *IM* reaches a fixpoint and returns $K_0 = K_{init} \wedge b \geq e \wedge f < a$.

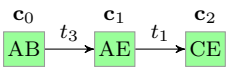


Figure 2: Trace set

For any parameter valuation satisfying K_0, the trace set (actually reduced to a single trace) is the one given in Figure 2. Beyond the application to robustness or optimization of some constants, the resulting constraint also gives valuable information about the relationship between the timing requirements. For example, c and d do not appear within K_0 (apart from $c \leq d$): this can be explained as follows. Initially, the token in A inhibits t_2; then, t_3 must fire first (because $f < a$) and, when t_1 finally fires (thus disinhibiting t_2), there is no token anymore in B; hence t_2 can never fire if $f < a$, whatever are the values of c and d.

3.3 Results

Theorem 1 (Correctness)**.** *Let $\mathcal{P}$ be a PITPN, and π_0 be a reference valuation. Let $K_0 = IM(\mathcal{P}, \pi_0)$. Then: (1) $\pi_0 \models K_0$ and (2) $\forall \pi \models K_0$, $\mathcal{P}[\pi]$ and $\mathcal{P}[\pi_0]$ have the same trace set.*

This result allows one to quantify the robustness of the system: the equality of trace sets implies that any linear-time (LTL) property that is true in $\mathcal{P}[\pi_0]$ is also true for $\mathcal{P}[\pi]$. Hence, if the correctness is given under the form of an LTL property, the timing requirements can safely vary as long as they satisfy K_0. Observe that timed properties (involving quantitative information) can also be modeled as properties on traces, by adding an observer to the system.

Due to the non-deterministic selection of an inequality, *IM* is non-confluent. As a consequence, it is also non-complete. Formally:

Proposition 1. *There may exist $\pi \not\models K_0$ such that $\mathcal{P}[\pi]$ and $\mathcal{P}[\pi_0]$ have the same trace set.*

Future Work. The inverse method is not guaranteed to terminate in the setting of parametric timed automata. However, the counter-example used in [ACEF09] cannot be used for PITPNs; proving (non-)termination is the subject of ongoing work. An implementation (for instance in IMITATOR [AFKS12]) should be performed. We also plan to extend the method to other classes of Petri nets, such as timed extensions of colored Petri nets [JK09]. Furthermore, this work could be combined with the modular state space exploration for timed Petri nets [LP07].

References

[ACEF09] Étienne André, Thomas Chatain, Emmanuelle Encrenaz, and Laurent Fribourg. An inverse method for parametric timed automata. *International Journal of Foundations of Computer Science*, 20(5):819–836, 2009.

[AFKS12] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *FM'12*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36, Paris, France, 2012. Springer.

[JK09] Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.

[LP07] Charles Lakos and Laure Petrucci. Modular state space exploration for timed Petri nets. *Journal of Software Tools for Technology Transfer*, 9(3-4):393–411, 2007.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[Mar11] Nicolas Markey. Robustness in real-time systems. In *SIES'11*, pages 28–34, Västerås, Sweden, 2011. IEEE Computer Society Press.

[TLR09] Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. Parametric model-checking of stopwatch Petri nets. *Journal of Universal Computer Science*, 15(17):3273–3304, 2009.

Multicore Timing Analysis using Abstract Interpretation and Model Checking

Mingsong Lv[1], Nan Guan[1], Wang Yi[1] and Ge Yu[2]

[1] Uppsala University, Uppsala, Sweden
[2] Northeastern University, Shenyang, China

Abstract

It is predicted that multicores will be increasingly used in future embedded real-time systems for high performance and low energy consumption. The major obstacle is that we may not provide any guarantee on real-time properties of software on such platforms due to inter-core conflicts. Estimating WCET becomes a key challenge on multicore architectures, since it is very difficult to model and analyze the behaviors on shared resources. In this paper, we study a multicore architecture where each core has local cache and all cores may access shared resources. We use Abstract Interpretation (AI) to analyze the local cache behavior of a program running on a dedicated core. Based on the cache analysis, we construct a Timed automaton (TA) for each program to model when the program accesses shared resources. We also model shared resources using timed automata. The TA models for the programs and shared resources will be explored using the UPPAAL model checker to find the WCETs for the respective programs. Based on the presented techniques, we have developed a tool (called McAiT) for multicore timing analysis. Experimental results on analyzing shared buses show significantly tightened WCET bounds compared with the worst-case bounds estimated based on worst-case bus access delay.

1 Background and Motivation

It is predicted that multicores will be increasingly used in future embedded real-time systems for high performance and low energy consumption. The major obstacle is that we may not predict or provide any guarantee on real-time properties of software on such platforms, because it is very hard to model and analyze the timing behavior of the programs on shared resources.

For example, the shared memory bus is such a resource that severely degrades the timing predictability of multicore software. In single-core WCET analysis, it is usually assumed that it takes constant time to access the main memory (given perfect memory controller). But for multicores, this assumption never holds, since memory accesses suffer extra delays as a result of access conflicts on the shared memory bus. For example, for a dual core processor with first-come-first-serve (FCFS) bus arbitration, the worst-case occurs when any memory access request is delayed by a request just issued from the other core. In this case, the worst-case memory access time will be doubled compared to that without bus contention. Since the number of processor cores on a chip continues to increase, the amount of traffic on the shared memory buses increases accordingly, and this problem is expected to be even worse in the future [1]. Similar problems exist on other shared resources such as shared caches. Since the time to access the shared resources is not trivial, assuming worst-case access delay will lead to very pessimistic WCET estimations. So techniques that can precisely bound the delays on shared resources are very important to obtain useful timing guarantees for multicore real-time systems.

In this paper, we study a multicore architecture where each core has a local L1 cache and all cores share the memory bus. We intend to find the timing bounds of the programs considering variable shared resource access delays due to resource contention. In order to precisely estimate

those delays, one has to analyze how programs' behaviors affect the timing of the conflicts, which is not a trivial task.

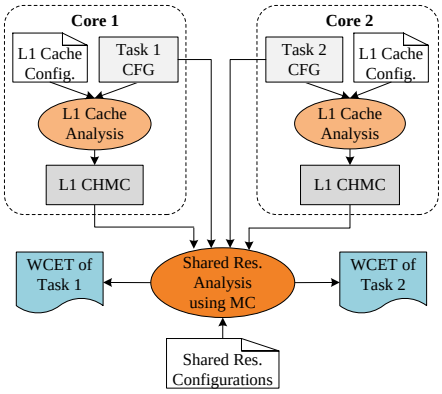


Figure 1: The overall analysis framework

2 Our Approach and the Tool

To solve the above timing analysis problem, we present a framework illustrated in Figure 1. The framework differs from exiting work in that we adopt abstract interpretation for cache analysis to improve analysis efficiency, the result of which is utilized in shared resource analysis by model checking to precisely capture the timing behaviors.

The analysis process consist of two major stages. In the first stage, we employs abstract interpretation (presented in [2]) to analyze local cache behaviors. Abstract Interpretation leverages three main fixed-point analyses over abstract cache domain to predict hits or misses:

- `MUST analysis`: to determine whether a memory access is "Always Hit (AH)";
- `MAY analysis`: to determine whether a memory access is "Always Miss (AM)";
- `PERSISTENCE analysis`: to determine whether a memory block is never evicted from the cache once loaded.

We call the results obtained by the first stage Cache Hit/Miss Classification (CHMC). This information is very useful since the access to shared resources (either last level cache or memory bus) is possible only when a local cache miss occurs. In multicores, shared resource access time may be very unpredictable since the accesses may suffer additional delay due to access conflicts on the shared resources. So in the second analysis stage, to precisely estimate the access delay, we use model checking to model the time at which the local cache misses occur and how conflicts happen on the shared resources. This involves two sub-tasks:

- `Modeling program behavior`: After AI analysis, we have obtained the the Control Flow Graph (CFG) of the program and the local cache behavior. We first build the sub-model for each basic block; then the TA model for the program is generated by connecting the sub-models according to the control flow. The TA model for each program precisely captures the timing behavior of the program's interaction with its environment, i.e., all the time sequences of the program for accessing the shared resources.

- `Modeling shared resource behavior`: The shared resources are also modelled with timed automata which simulate their behavior. For example, to model an FCFS bus, we need to model in the TA how the bus accesses are queued and how long each access takes; to model a shared cache, the TA needs to maintain the cache contents and simulate the cache replacement policy.

The TA models for the programs and the shared resources form a network of timed automata. Then we can let the UPPAAL model checker [3] to explore the TA models, and the WCET of a program is extracted from the clock constraints within UPPAAL. Our proposed framework can produce very tight WCET estimations. Experimental results for the analysis of the FCFS bus show that the WCET bounds can be tightened by up to 240% compared with the worst-case bounds estimated based on worst-case bus access delay. We refer interested readers to our recent work [4] for more details on the results.

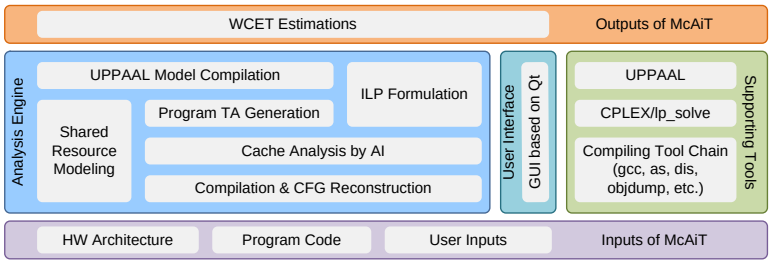


Figure 2: The architecture of McAiT

Based on the techniques presented, we have developed a WCET tool called McAiT [5] for analysis of multicore software, which is illustrated in Figure 2. The tool consists of an analysis engine which implements lots of WCET analysis techniques, a GUI that provides functionalities to manage the analysis procedures, and supporting tools such as the UPPAAL model checker, lp_solve, etc. The analysis framework presented above is implemented in the analysis engine. McAiT also supports the classical Implicit Path Enumeration technique combined with worst-case resource access delay for WCET estimation, to provide the users with the flexibility to trade analysis precision for efficiency. The details for the tool are reported in [5].

References

[1] S. Williams, A. Waterman, and D. Patterson. Roofline: an Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76, 2009.

[2] C. Ferdinand. Cache Behavior Prediction for Real-Time Systems. *Ph.D. thesis of Saarland University*, 1997.

[3] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. Springer, 2004.

[4] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In RTSS 2010.

[5] Mingsong Lv, Nan Guan, Wang Yi, and Ge Yu. McAiT – A Timing Analyzer for Multicore Real-Time Software. In ATVA 2011.

A Novel Cache PERSISTENCE Analysis for Worst-Case Execution Time Estimation

Mingsong Lv[1], Nan Guan[1], Wang Yi[1] and Ge Yu[2]

[1] Uppsala University, Uppsala, Sweden
[2] Northeastern University, Shenyang, China

Abstract

Cache analysis for WCET estimation requires PERSISTENCE analysis which predicts whether a memory block is never evicted from the cache once it is loaded. Ferdinand's PERSISTENCE analysis based on abstract interpretation has been dominating for years, but it was found to be unsafe recently. Two most recent methods have been proposed by Cullmann and Huynh respectively to correct the safety problem in Ferdinand's approach, but they share a common problem of severe overestimation of the maximal ages for the memory blocks. In this paper we present a new PERSISTENCE analysis, which is proved to be not only safe, but also as precise as either Cullmann's approach or Huynh's approach. We also demonstrated that our new analysis technique does not suffer the problems that make Cullmann or Huynh's approach imprecise.

1 Background and Motivation

Hard real-time systems require that the timing behavior of the system is analyzed off-line to guarantee the timing constraints are met at run time. One of the key tasks to achieve this is Worst-Case Execution Time (WCET) analysis [1], which derives an upper bound for the execution time of a program. The upper bounds are typically required to be as tight as possible so that over-provisioning of system resources is avoided. To enforce this, WCET analysis must take cache into consideration since it significantly affects the execution time of a program [2]. In modern processors, if a data reference is hit in the cache, it consumes only several CPU cycles; otherwise, the processor has to take hundreds of CPU cycles to fetch the required data from off-chip main memory. However, determining whether each memory access is a hit or a miss in the cache is a challenging problem, which has been a hot research theme for many years.

In the past decades, cache analysis has been dominated by "Abstract Interpretation (AI) [3]". The AI framework leverages two main fixed-point analyses over abstract cache domain to predict hits or misses:

- `MUST analysis`: to determine whether a memory access is "Always Hit (AH)";
- `MAY analysis`: to determine whether a memory access is "Always Miss (AM)".

In fact, in real-life programs, most of the execution time is spent in loops. Due to the "locality principle", most of the memory accesses within loops exhibit such a typical characteristic: once a memory block is loaded into the cache, it is never evicted as long as the program is in the loop. It is clear that such memory access behavior cannot be classified as either AH or AM, so cache PERSISTENCE analysis was proposed to capture such behavior. With PERSISTENCE analysis, the precision of predicting cache hits or misses is greatly improved.

Traditionally, PERSISTENCE analysis is mainly used to identify memory blocks that are guaranteed to be in the cache once loaded, and people do not really care about the exact "age" of the memory blocks in the cache. However, the most recent research in timing analysis of

real-time systems calls for a strong requirement of predicting the maximal ages. For example, in multicore WCET analysis [4], one must estimate the maximal age of a memory block to know how many conflicts it can suffer before evicted by the execution of other concurrent programs. Another major usage of predicting maximal age exists in cache analysis for the MRU replacement [5], in which the number of cache misses for each memory block is directly related to the maximal age predicted under LRU replacement. Unfortunately, existing PERSISTENCE analysis techniques cannot produce tight estimations on the maximal ages of memory blocks.

2 Related Work and Main Problems

The most widely adopted PERSISTENCE analysis technique is proposed by Ferdinand [6] (which is based on the Abstract Interpretation (AI) theory), since it is demonstrated to be both precise and efficient. But 12 years after Ferdinand's analysis was proposed, it has been found to be unsafe [7, 8]: the ages of the memory blocks can be underestimated due to the incorrect cache update semantics. Since "safety" is the foremost consideration in the design of hard real-time systems, this problem drew lots of attention from different research groups. Cullmann presented an approach to correct Ferdinand's error [7], which adopts a new cache update function to guarantee safety and furthermore leverages MAY analysis to remove some pessimism in the estimations. Huynh et al. also proposed a safe PERSISTENCE analysis [8], which maintains a "Younger Set" for each memory block to correctly estimate its age. However, these two state-of-the-art techniques suffer precision problems w.r.t. the ages predicted.

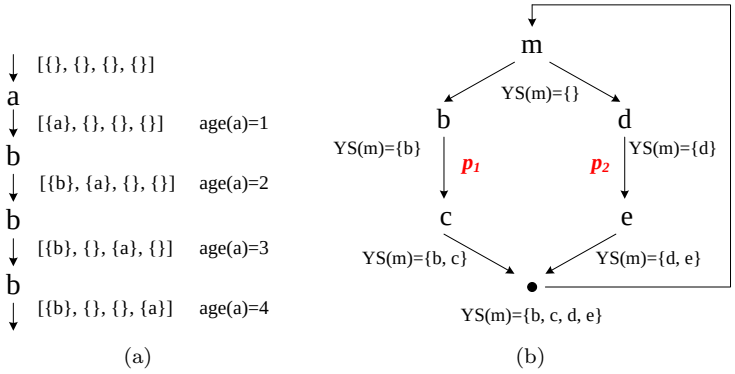


Figure 1: Examples to show the problem of Cullmann's approach and Huynh's approach

In order to correct the safety problem in Ferdinand's approach, Cullmann adopted such a principle in updating the abstract cache state (ACS): the access of any memory block will cause all the other blocks in the ACS to age by 1, excluding the blocks in the last cache line in cases where the total number of blocks in the ACS is not bigger than the cache associativity. Cullmann's approach is imprecise when consecutive accesses to the same block occur. For example in Figure 1(a), consider a memory access sequence "*abbb*", according to Cullmann's PERSISTENCE update semantics, the maximal age for block a is 4; while it is clear that all the 3 accesses to b can only age block a by 1. The problem is common in real-life systems since a memory block typically contains multiple instructions, the execution of which generates consecutive accesses to the same memory block.

Huynh's approach maintains a "Younger Set" for each memory block with the objective to

correct the safety problem in Ferdinand's approach. Unfortunately, this approach also suffers precision problems. Consider an example illustrated in Figure 1(b), according to Huynh's approach, the final Younger Set for memory block m is $\mathcal{YS}(m) = \{b, c, d, e\}$, so the maximal age for m is 5. But a closer look at the program shows that no matter how the execution switches between path "p_1" or "p_2", it must first access block m, so the actual maximal age of m is only 3. That is to say, block b and d are equivalent in aging block a (similar for c and e). Huynh's approach are not able to find this property, and it works worse in programs where lots of branching structures exist.

3 Our Approach

To tighten the maximal ages predicted, we presented a new approach for PERSISTENCE analysis, which works as follows: we still adopt the fixed point iteration analysis framework used in AI analysis and we conduct MAY analysis and PERSISTENCE analysis in parallel. At any point the PERSISTENCE ACS is updated, we calculate a range for the age of each memory block, with the lower bound extracted from MAY ACS and the upper bound extracted from the PERSISTENCE ACS. Then a function is invoked which can efficiently decide whether the access to the current block will cause the blocks with older ages to age by 1. By this distinction, pessimistic aging of memory blocks is avoided.

Due to limited space, we do not present the details of our approach here. We have proved that our proposed approach is safe with respect to predicting the maximal ages for memory blocks. We also strictly proved that our proposed approach is at least as precise as either Cullmann's approach or Huynh's approach, and demonstrated that our approach does not suffer the consecutive access sequence problem (which kills Cullmann's approach) or the problem in dealing with branches (which kills Huynh's approach), i.e., our approach can produce better results in predicting the maximal ages in PERSISTENCE analysis.

References

[1] Reinhard Wilhelm, and et. al. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. ACM Trans. Embed. Comput. Syst., 2008.

[2] Reinhard Wilhelm, Daniel Grund, Jan Reineke, and et al. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. IEEE Transaction on CAD of Integrated Circuits and Systems, 28(7): 966-978, 2009.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 238-252, 1977.

[4] Sudipta Chattopadhyay, and et al. A Unified WCET Analysis Framework for Multi-core Platforms. IEEE Real-Time and Embedded Technology and Applications Symposium, 99-108, 2012.

[5] Nan Guan, Mingsong Lv, Wang Yi, Ge Yu. WCET Analysis with MRU Caches: Challenging LRU for Predictability. In Real-Time and Embedded Technology and Applications Symposium, 2012.

[6] Christian Ferdinand. Cache behavior prediction for real-time systems. PhD thesis, Saarland University, 1999.

[7] Christoph Cullmann. Cache persistence analysis: a novel approach, theory and practice. Proceedings of the ACM Conference on Languages, compilers, and tools for embedded systems, 2011.

[8] Bach Khoa Huynh, Lei Ju, Abhik Roychoudhury. Scope-Aware Data Cache Analysis for WCET Estimation. IEEE Real-Time and Embedded Technology and Applications Symposium, 2011.

Semi Inversion of Conditional Constructor Term Rewriting Systems

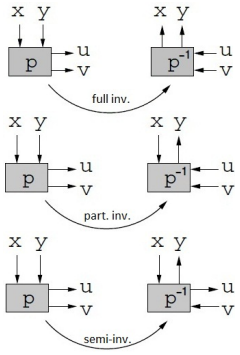
M. Tønnesen and R. Glück

DIKU, Dept. of Computer Science, University of Copenhagen, maja@tonnesen.org, glueck@acm.org

1 Introduction

Programs that are inverse to each other are practical and familiar to everyone, such as the encoding and decoding of data by the programs zip and unzip. The transformation of a zip-program into an unzip-program is known as *full inversion*. *Semi-inversion*, the most general form of inversion, transforms a relation into a new relation that takes some of the original inputs and outputs as new input. For example, the inversion of a symmetric encrypter into a decrypter cannot be done by full inversion because both programs take the same key as input.

We present an algorithm for the semi-inversion of deterministic *conditional constructor term rewriting system* (CCS) [13]that generates inverses of the same or an expanded class of systems. The new algorithm is more general than existing inversion algorithms and produces in many cases better results. An implementation in Prolog has been applied to both semi- and full-inversion problems including the automatic inversion of a simple encrypter and of an interpreter for a reversible programming language. The latter is a self-inverse program and our algorithm successfully inverts it into a copy of itself modulo var. renaming.



We distinguish between three forms of program inversion: A *full inversion* turns p into a new program $\mathtt{p}^{-1}$ where all of the original inputs and outputs are exchanged. If p is injective then $\mathtt{p}^{-1}$ implements a function. A *partial inversion* yields a program $\mathtt{p}^{-1}$ that inputs the original outputs (u, v) and some of the original inputs (x), and returns the remaining input (y). A *semi-inversion* yields a program $\mathtt{p}^{-1}$, which given some of the original inputs (x) and some of the original outputs (v) returns the remaining input (y) and output (u). The programs p and $\mathtt{p}^{-1}$ may implement functions or more generally relations. Full inversion is a subproblem of partial inversion, which is a subproblem of semi-inversion.

Dijkstra studied the full inversion of programs by hand in a guarded commands language [3]. Following, a few program inversion algorithms have been developed, and then often for different languages and different types of inversion. Of those, only Nishida et al. [11,12] and Almendros-Jiménez et al. [1] have considered term rewriting systems, where the latter constrained the systems such that terms have only one normal form. Mogensen [8,10], who developed the first semi-inversion algorithm, did so for a deterministic guarded equational language. Algorithms for full inversion were studied in the context of functional languages [4–7].

A main advantage of using conditional constructor term rewriting systems for program inversion is that the semantics of functions and relations can be expressed and efficiently calculated in the same formalism. Our algorithm is more general than existing inversion algorithms for functional languages and rewriting systems, and produces in many cases better results. The following table compares our algorithm with other existing inversion algorithms:

	functions	relations
full inversion	Glück and Kawabe [5]	Nishida [11]
partial inversion	Almendros-Jiménez et al. [1]	Nishida et al. [12]
semi-inversion	Mogensen [8]	*This algorithm*

The contribution of this work is (1) a semi-inversion algorithm for CCS which returns a CCS which is a semi-inversion with respect to a function, known part of the input and known part of the output, or it returns a CCS with unrestricted variables and a warning of this. (2) given an heuristic which in many cases minimizes the search space (3) implemented and applied the algorithm to an interpreter for a reversible language, a small encryption algorithm, and the falling object example.

2 The Semi-Inversion Algorithm—An Outline

The semi-inversion algorithm has two main phases:

1. *Preprocessing* transforms unconditional constructor rules into *flat conditional constructor rules*, that is, rewrite rules with conditions and not containing nested function symbols, and adds new rules to ensure the equivalence of the set of rewritings of the original and the preprocessed rules.
2. *Inversion* labels all function symbols with two index sets, I and O, that contain the indices of the known variables of the left-hand and right-hand side of a rule, respectively, and then locally inverts every rule, such that all known variables given by the index sets occur in the left-hand side and the rest on the right-hand side of the new rule.

The main loop of the semi-inversion algorithm controls the application of the preprocessing and inversion to rewrite rules. It keeps track of the functions (defined symbols) that have been semi-inverted and those pending. For each of the pending functions it applies preprocessing and inversion to each of the defining rules. Afterwards, the pending tasks are updated by removing the inverted function and adding any new dependencies. When no pending functions exists, it returns a CCS containing the semi inverted function and all its dependencies.

3 Results

We illustrate the semi-inversion algorithm with three examples:

- **Discrete simulation of free fall** Consider a discrete simulation of an object that falls through a vacuum [2]. The object has two associated variables, h_t and v_t, giving its height in meters and downward velocity in meters per second at time t, respectively. Initially, $v_0 = 0$ and h_0 is the height from which the object is dropped. The object has two associated two associated variables, h_t and v_t, giving its height in meters and downward velocity in meters per second at time t, respectively. Initially, $v_0 = 0$ and h_0 is the height from which the object is dropped. In the original function, we input start height and velocity and the amount of time the object falls and calculate height and velocity after the time has passed. We have succesfully produced 4 semi-inversions of the original program shown in the table. The `semi-inversion #2` cannot be created by any of the algorithms presented by $[1, 5, 8, 11, 12]$.

`Original:`	$(\mathrm{h_{start}}, \mathrm{v_{start}}, \mathrm{t}) \rightarrow (\mathrm{h_{end}}, \mathrm{v_{end}})$
`Full inversion:`	$(\mathrm{h_{end}}, \mathrm{v_{end}}) \rightarrow (\mathrm{h_{start}}, \mathrm{v_{start}}, \mathrm{t})$
`Partial inversion:`	$(\mathrm{h_{end}}, \mathrm{v_{end}}, \mathrm{t}) \rightarrow (\mathrm{h_{start}}, \mathrm{v_{start}})$
`Semi-inversion #1:`	$(\mathrm{v_{start}}, \mathrm{t}, \mathrm{h_{end}}) \rightarrow (\mathrm{h_{start}}, \mathrm{v_{end}})$
`Semi-inversion #2:`	$(\mathrm{v_{start}}, \mathrm{t}, \mathrm{v_{end}}) \rightarrow (\mathrm{h_{start}}, \mathrm{h_{end}})$

- **Encrypter and decrypter** Automatic generation of decrypters are fascinating, and is on purpose very hard in many cases. The following encryption is a modification of a simple encryption method presented in [9]. It produces an encypted text `z:zs` given a text formed as a list `x:xs` and a key `key`. Encryption cleans the key by mod 4, adds the new value to the first character, and repeats the process recursively for the rest of the text (the modification is that we use mod 4 instead of mod 256, since mod 256 consists of 257 rules). The encrypter is given to the left and the decrypter

is a partial inversion of it with respect to 2nd input {2} and the output {1} is shown to the right. The decrypter ($\overline{\texttt{encrypt}}_{\{2\}\{1\}}$) produces the decrypted text `x:xs` given the key `key` and the encrypted text `z:zs`. Note that `mod4` is equivalent to $\overline{\texttt{mod4}}_{\{1\}\emptyset}$.

Original encrypter:

```
encrypt(nil,key) → ⟨nil⟩
encrypt(x:xs,key) → ⟨z:zs⟩ ⇐
        mod4(key) → ⟨y⟩,
        add(x,y) → ⟨z⟩,
        encrypt(xs,key) → ⟨zs⟩
mod4(0) → ⟨0⟩
mod4(s(0)) → ⟨s(0)⟩
mod4(s²(0)) → ⟨s²(0)⟩
mod4(s³(0)) → ⟨s³(0)⟩
mod4(s⁴(x))→ ⟨w0⟩ ⇐  mod4(x)→ ⟨w0⟩
```

The generated decrypter:

```
encrypt_{2}{1}(key,nil)→ ⟨nil⟩
encrypt_{2}{1}(key,z:zs) →  ⟨x:xs⟩ ⇐
        mod4_{1}∅(key) → ⟨y⟩,
        add_{2}{1}(y,z) → ⟨x⟩,
        encrypt_{2}{1}(key,zs) → ⟨xs⟩
mod4_{1}∅(0)→ ⟨0⟩
mod4_{1}∅(s(0))→ ⟨s(0)⟩
mod4_{1}∅(s²(0))→ ⟨s²(0)⟩
mod4_{1}∅(s³(0))→ ⟨s³(0)⟩
mod4_{1}∅(s⁴(x))→ ⟨w0⟩ ⇐ mod4_{1}∅(x)→ ⟨w0⟩
mod4_{1}∅(w0) → ⟨udef(w0)⟩
```

- **Inverted inverter** The Janus language is a reversible language [14] where functions can be both called (executed in a forward direction) and uncalled (executed in a backward direction). An inverter for the Janus language creates procedures which are equal to uncalling the original procedure. Since Janus is a reversible language the full inversion of the Janus-inverter is equivalent to itself. We have implemented a subset of the Janus-inverter in a CCS, and created its full-inversion by applying the semi-inversion algorithm.

References

[1] J. M. Almendros-Jiménez, G. Vidal. Automatic partial inversion of inductively sequential functions. In *IFL*, *LNCS*, Vol. 4449, 253–270. Springer, 2006.

[2] H. B. Axelsen, R. Glück, T. Yokoyama. Reversible machine code and its abstract processor architecture. In *Comp. Sc. – Theory and Applications*, *LNCS*, Vol. 4649, 56–69. Springer, 2007.

[3] E. W. Dijkstra. Program inversion. In *Program Construction*, *LNCS*, Vol. 69, 54–57. Springer, 1978.

[4] D. Eppstein. A heuristic approach to program inversion. In *IJCAI*, 219–221, 1985.

[5] R. Glück, M. Kawabe. A program inverter for a functional language with equality and constructors. In *Prog. Lang. and Systems. Proceedings*, *LNCS*, Vol. 2895, 246–264. Springer, 2003.

[6] R. Glück, M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundamenta Informaticae*, 66:367–395, 2005.

[7] R. Glück, M. Kawabe. Revisiting an automatic program inverter for Lisp. *SIGPLAN Notices*, 40(5):8–1, 2005.

[8] T. Æ. Mogensen. Semi-inversion of guarded equations, *GPCE, LNCS*. Vol. 3676, 189–204. Springer, 2005.

[9] T. Æ. Mogensen. Report on an implementation of a semi-inverter. In *Perspectives of Systems Informatics*, *LNCS*, Vol. 4378, 322–334. Springer, 2007.

[10] T. Æ. Mogensen. Semi-inversion of functional parameters. In *PEPM*, 21–29. ACM, 2008.

[11] N. Nishida. *Transformal Approach to Inverse Computation in Term Rewriting*. PhD thesis, Graduate School of Engineering, Nagoya University, 2004.

[12] N. Nishida, M. Sakai, T. Sakabe. Partial inversion of constructor term rewriting systems. In *RTA*, *LNCS*, Vol. 3467, 264–278. Springer, 2005.

[13] E. Ohlebusch. Transforming conditional rewrite systems with extra variables into unconditional systems. In *LPAR*, *LNCS*, Vol. 1705, 111–130. Springer, 1999.

[14] T. Yokoyama, H. B. Axelsen, R. Glück. Principles of a reversible programming language. In *Conference on Computing Frontiers. Proceedings*, 43–54. ACM Press, 2008.

Certified CYK parsing of context-free languages

Denis Firsov and Tarmo Uustalu

Institute of Cybernetics at TUT, Tallinn, Estonia, {denis,tarmo}@cs.ioc.ee

Abstract

We describe our ongoing work on certified parsing of context-free languages. We have implemented the functional core of the Cocke-Younger-Kasami (CYK) parsing algorithm (based on multiplication of matrices over the powerset of the non-terminals of a normal-form grammar) and proved its correctness in the Agda dependently typed programming language.

1 Introduction

In our previous work [2] we implemented a certified parser-generator for regular expressions based on a matrix representation of finite-state automata using the dependently typed programming language Agda [1]. Now we want to show how the same can be done for a wider class of languages.

We decided to implement the Cocke–Younger–Kasami (CYK) parsing algorithm for context-free grammars [4] because of its elegant structure. This algorithm is based on multiplication of matrices over the powerset of non-terminals. Valiant [3] showed how to modify CYK algorithm so as to use Boolean matrix multiplication. We avoid this approach because of the details of the low-level encoding that obfuscate the higher-level structure of the algorithm.

2 CYK and Its Correctness

2.1 Context free grammars and matrices

Before giving the algorithm we need to define the form of grammars we work with and some matrix operations.

Definition 2.1. *A context-free grammar is a 5-tuple* $(N, \Sigma, P, S, nullable)$ *with*

1. *N: The set of nonterminals*
2. Σ*: The set of terminals*
3. *P: The set of productions, each of which has the form:*
   $A \longrightarrow BC$ *or* $A \longrightarrow x$*, for* $x \in \Sigma$ *and* $A, B, C \in N$
4. *S: The starting nonterminal*
5. *nullable: A boolean flag telling whether the empty word is part of the language or not*

Definition 2.2. *Let* a *and* b *be two* $n \times n$ *matrices with subsets of* N *as elements. Then their multiplicaton* $c = a * b$ *is defined as:*

$$c_{i,j} = \bigcup_{k=1}^{n} a_{i,k} * b_{k,j} \quad \forall 1 \leq i, j \leq n$$

where multiplication of subsets of N *is defined as:*

$$N_1 * N_2 = \{A \mid B \in N_1,\ C \in N_2,\ (A \longrightarrow BC) \in P\}$$

The sum of two matrices $c = a \cup b$ is defined pointwise by

$$c_{i,j} = a_{i,j} \cup b_{i,j} \qquad \forall 1 \leq i, j \leq n \tag{1}$$

Definition 2.3. *Let b be an $n \times n$ matrix. Then taking b into the m-th power, denoted by $b^{(m)}$, is defined by*

$$b^{(m)} = \bigcup_{j=1}^{m-1} b^{(j)} * b^{(m-j)} \qquad \text{and} \qquad b^{(1)} = b \tag{2}$$

2.2 The algorithm

The CYK parsing algorithm takes a grammar G and a string $w = w_1 \dots w_n$ and checks if w can be derived from the starting nonterminal S of G. The algorithm works as follows:

1. Define a $(n+1)\times(n+1)$ matrix b by $b_{i,j} = \emptyset$ for $j \neq i+1$ and $b_{i,i+1} = \{A \mid (A \longrightarrow w_i) \in P\}$
2. Compute the matrices $b^{(2)}, \dots, b^{(n)}$
3. Check whether $S \in b^{(n)}_{1,n+1}$

First, the parses of all substrings of w of length 1 are recorded in the matrix b (step 2). Then the matrices $b^{(m)}$ are computed. They give us the parses of all substrings of w of length at most m (step 3). The main invariant of computation is:

$$A \in b^{(m)}_{i,i+m} \iff A \longrightarrow^* w_i \dots w_{i+m-1}$$

2.3 Correctness

We will now sketch the correctness statements and proofs of the algorithm, structured in a formalizable fashion.

To formulate the correctness statements we first define the parses of substrings of a given string. They can be given by a 4-place relation, denoted by $A \Longrightarrow w[i][j]$, which states that the substring of w from the i-th to the $(j-1)$-th position is derivable from the nonterminal A.

Definition 2.4. *The parse relation is defined inductively by*

1. *If nullable = true, then $S \Longrightarrow w[1][1]$*
2. *If $(A \longrightarrow w_i) \in P$, then $A \Longrightarrow w[i][i+1]$*
3. *If $B \Longrightarrow w[i][k]$, $C \Longrightarrow w[k][j]$ and $(A \longrightarrow BC) \in P$, then $A \Longrightarrow w[i][j]$*

Correctness consists of soundness and completeness. We start by soundness.

Theorem 2.5. *If $A \in b^{(m)}_{i,j}$ then $A \Longrightarrow w[i][j]$, where b is the $(n+1)\times(n+1)$ matrix constructed according to the CYK parsing algorithm.*

Proof. The proof is by complete induction on m. The base case is immediate.

From $A \in b^{(m)}_{i,j}$ we know that for some k and l we have $A \in (b^{(k)} * b^{(l)})_{i,j}$.

Next, from the definition of matrix multiplication we know that there exist B, C and z such that $B \in b^{(k)}_{i,z}$, $C \in b^{(l)}_{z,j}$ and $(A \longrightarrow BC) \in P$.

By the induction hypothesis we get $B \Longrightarrow w[i][z]$ and $C \Longrightarrow w[z][j]$. Finally, by the definition of the parsing relation we conclude $A \Longrightarrow w[i][j]$. □

Next, we show that the parsing algorithm is complete.

Theorem 2.6. *If $A \Longrightarrow w[i][i+m]$ then $A \in b^{(m)}_{i,i+m}$, where b is the $(n+1) \times (n+1)$ matrix constructed according to the CYK parsing algorithm.*

Proof. The proof is by induction on the parsing derivation.

The first two base cases are immediate from the definitions.

In the third case we know that there exist B, C and k such that $B \Longrightarrow w[i][k]$, $C \Longrightarrow w[k][i+m]$ and $(A \longrightarrow BC) \in P$.

Observe that $k = i + c$ for some c and $i + m = i + c + d$, for some d.

By the induction hypothesis $B \in b^{(c)}_{i,i+c}$ and $C \in b^{(d)}_{i+c,i+c+d}$. Hence, from the definition of multiplication and $(A \longrightarrow BC) \in P$ we conclude that $A \in (b^{(c)} * b^{(d)})_{i,i+c+d}$.

From $i + m = i + c + d$ we get $m = c + d$, which yields $b^{(c)} * b^{(d)} \subseteq b^{(m)}$. □

Our proofs of correctness make explicit the induction principles employed and other details which are usually left implicit in formal languages literature.

In our Agda development, we have programmed the algorithm together with the soundness and completeness proofs just shown. The most interesting aspect is the design of the datatypes employed by the algorithm in such a way that the relevant invariants are enforced.

Our implementation of the algorithm is structurally recursive on the length n of the given string w. Without the tabulation that is the key of the effiency of the imperative algorithm, the functional version involves excessive redundant recomputation of the matrices $b^{(m)}$. We intend to program the tabulation (memoization) too, and prove it correct, but want to keep the correctness arguments of the core of the algorithm and the memoization separate.

3 Conclusion

Our experiment reproves it again that dependently typed programming languages are a powerful tool for coding non-trivial algorithms while also proving them correct.

Acknowledgements This work was supported by the Estonian Centre of Excellence in Computer Science, EXCS, a project funded by the European Regional Development Fund, ERDF, and the target-financed research theme no. 0140007s12 of the Estonian Ministry of Education and Research.

References

[1] Agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`, 2012.

[2] Denis Firsov. Certified parsing of regular languages. Master's thesis, Tallinn University of Technology, 2012.

[3] Leslie G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–314, April 1975.

[4] D.H. Younger. Recognition and parsing of context-free languages in time n^3. *Inf. Control*, 10(2):189–208, 1967.

Tetrasystems: A Framework for String Generation Devices

Eero Lassila

Aalto University, Helsinki, Finland
eero.lassila@aalto.fi

Abstract

Tetrasystems are proposed as a common framework for practical string generation devices such as macro processors and parametric Lindenmayer systems. They are also capable of emulating many devices well-known from formal language theory.

1 Introduction

We are especially interested in rewriting systems neither as language recognition devices nor as language generation devices but as string generation devices, like macro processors [1] or parametric Lindenmayer systems [11, 10, 9], which are proven programming tools. Therefore, we want to allow infinite alphabets (for coping with structured symbols) and unbounded context-sensitivity. (For our earlier attempts in this direction, see [5, 4].)

We propose tetrasystems as a common framework for such string generation devices. However, we also claim that tetrasystems are able to emulate large families of devices well-known from formal language theory. This notion of emulation will be explicated in the actual NWPT 2012 presentation.

Section 2 contains preliminary definitions, and section 3 then introduces tetrasystems. Finally, section 4 takes a closer look at tetrasystem-based emulation of other devices.

2 Preliminary Definitions

The trees considered here are finite, rooted, and ordered. Each tree node moreover holds a *letter*. The letters divide into *nonterminals* and *terminals*. *Words* are finite letter sequences.

2.1 Span Between Two Tree Nodes

Let n_1 and n_2 be two given nodes in a given tree, and let n_0 denote the lowest (i.e. closest) common ancestor of n_1 and n_2. We define the *span* between n_1 and n_2 as the following integer triple $\langle i, d, j \rangle$:

$$\begin{array}{rcl} i & = & \text{the difference of the depths of } n_1 \text{ and } n_0 \\ d & = & \begin{cases} -1 & \text{if } n_2 \text{ is on the left of } n_1 \\ 0 & \text{if (at least) one of } n_1 \text{ and } n_2 \text{ is an ancestor of the other} \\ 1 & \text{if } n_2 \text{ is on the right of } n_1 \end{cases} \\ j & = & \text{the difference of the depths of } n_2 \text{ and } n_0 \end{array}$$

(Of course, the depth of the root is 0, and the depth of a child is always one more than the depth of the parent.)

Let the span between n_1 and n_2 be denoted as $\angle(n_1, n_2)$. Obviously, $\angle(n_1, n_2) = \langle i, d, j \rangle$ implies $\angle(n_2, n_1) = \langle j, -d, i \rangle$; and trivially, $\angle(n_1, n_1) = \angle(n_2, n_2) = \langle 0, 0, 0 \rangle$.

2.2 Tree Belts and Belt-Selectors

A *belt* of a tree is simply a cross section of the tree, that is, such a subset of the tree nodes that contains exactly one ancestor of each leaf of the tree.

A *comb* is defined as any such function $f : \{\ldots, -2, -1, 0, 1, 2, \ldots\} \to \{0, 1, 2, \ldots\} \cup \{\infty\}$ that $\forall i \neq 0 : f(i) > 0$.

A *belt-selector* is a function that takes a tree and one of its nodes as its arguments and returns such a belt of the argument tree that contains no proper ancestor of the argument node. Moreover, there is a bijection between combs and belt-selectors, and so for any belt-selector s, there is a unique comb f_s.

But we still have to define how a belt-selector selects its belt, and to characterize the claimed bijection. For a given belt-selector s, for a given tree X, and for a given node n of X, the belt selected is the node set constructed by the following two successive steps:

1. Each such node $n' \in X$ that is not a proper ancestor of n is added to the set if $\angle(n, n') = \langle i, d, f_s(i \times d)\rangle$ for some i and d.

2. Each such leaf of X that has no ancestor already in the set is added to the set.

A belt-selector is said to be *settled* at a tree node if every leaf added by step 2 above holds a terminal. For example, consider the belt-selector s with $\forall i : f_s(i) = \infty$. This belt-selector is settled at a tree node if and only if all the leaves of the tree hold terminals. On the other hand, the belt-selector s^* with $\forall i : f_{s^*}(i) = \min(|i|, 1)$ is settled at every node in every tree.

2.3 Letter-Refiners

A *letter-refiner* is a function that takes three arguments: a word, a letter, and a word. The former word represents the left-hand context, and the latter the right-hand context. The letter-refiner returns a non-empty set of non-empty words, which represent the possible (mutually alternative) refinement results of the argument letter in the specified two-sided context. (Each terminal refines only to itself.)

3 Tetrasystems

A *tetrasystem* implements rewriting as a tree generation process. The operation of a tetrasystem is governed by a refinement rule base and a separate control mechanism. These two are intended to be as orthogonal to each other as possible.

The alphabet of a tetrasystem may be countably infinite, and the effective rewriting context may be unbounded in both directions.

3.1 Components of a Tetrasystem

The two main components of a tetrasystem are a letter-refiner (i.e. the rule base) and a *frame* (i.e. the control mechanism) consisting of four belt-selectors.

All the components of a given tetrasystem $\langle V_N, V_T, c_S, r, \langle s_1, s_2, s_3, s_4\rangle\rangle$ are as follows:

- set V_N of nonterminals, which must be non-empty but may be finite or countably infinite

- set V_T of terminals, which may be empty, finite, or countably infinite

- the *seed-letter* c_S must belong to V_N
- the letter-refiner r must have the property that every possible refinement result of each letter in V_N may contain only letters in $V_N \cup V_T$
- the frame $\langle s_1, s_2, s_3, s_4 \rangle$

3.2 Tetrasystem Operation

The tree generation process proceeds as follows:

1. The tree is booted up by introducing a single root node holding the seed-letter.
2. The tree grows by repeated expansion of leaves holding nonterminals:
   (a) (Use of s_1.) A given nonterminal-lettered leaf node is *fertile*, i.e. ready to be expanded, if the generation process has already proceeded sufficiently far in the other parts of the tree. That is, s_1 must be settled at the leaf.
   (b) (Use of s_2.) At a time, exactly one of the fertile leaves (if any) is expanded by applying the letter-refiner: the two sides of the refinement context are given by the belt returned by s_2; and the node changes from a leaf into a non-leaf as it is now provided with a child node sequence collectively holding one of the possible refinement results.
3. Output words (if any) of the process can be found at any time (and so even if the process has not terminated):
   (a) (Use of s_3.) A node is *mature* if it is nonterminal-lettered and s_3 is settled at it.
   (b) (Use of s_4.) For any mature node, the belt returned by s_4 gives an output word.

So the process does not terminate as long as there are fertile nonterminal-lettered leaves.

4 Emulation of Other Devices

We claim that tetrasystems are intrinsically capable of emulating macro processors and parametric Lindenmayer systems: one just has to choose the appropriate frame.

We also claim that choosing a suitable frame even, in a sense, enables the emulation of

1. FPIL-type Lindenmayer systems [13, 3],
2. context-sensitive Chomsky grammars [7], and
3. context-sensitive pure grammars [8, 6].

However, there are further restrictions concerning the empty word: in cases 2 and 3, the native production rules must never refine any letter into an empty word; and in cases 1 and 3, the native start words must not include the empty word.

We demonstrate the above claims by using *selective substitution grammars* [12, 2] simply for their original purpose: as a unifying rewriting framework, now for all the devices mentioned above (other than tetrasystems). This common intermediate representation facilitates the explication of our notion of emulation in the actual NWPT 2012 presentation.

The presentation will also precisely specify the frames needed in each one of the emulation cases mentioned above.

References

[1] Alfred J. Cole. *Macro Processors*. Cambridge University Press, 2nd edition, 1981.

[2] Jürgen Dassow and Gheorghe Păun. *Regulated Rewriting in Formal Language Theory*, chapter 10, pages 279–289. Springer, 1989.

[3] Lila Kari, Grzegorz Rozenberg, and Arto Salomaa. L systems. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 253–328. Springer, 1997.

[4] Eero Lassila. On tree belts and belt-selectors. In Nisse Husberg, Tomi Janhunen, and Ilkka Niemelä, editors, Leksa notes in computer science, pages 47–58. Technical Report HUT-TCS-A63, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, October 2000. ISBN 951-22-5211-2. `http://users.ics.aalto.fi/ela/publications/tr-tbb2000.pdf`.

[5] Eero Lassila. A tree expansion formalism for generative string rewriting. Technical Report HUT-TCS-B20, Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland, June 2001. ISBN 951-22-5554-5. `http://users.ics.aalto.fi/ela/publications/tr-tefgsr2001.pdf`.

[6] Alexandru Mateescu and Arto Salomaa. Aspects of classical language theory (section 7.1). In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 175–251. Springer, 1997.

[7] Alexandru Mateescu and Arto Salomaa. Formal languages: an introduction and a synopsis (section 3.1). In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 1, pages 1–39. Springer, 1997.

[8] Hermann A. Maurer, Arto Salomaa, and Derick Wood. Pure grammars. *Information and Control*, 44(1):47–72, 1980.

[9] Przemyslaw Prusinkiewicz. Simulation modeling of plants and plant ecosystems. *Communications of the ACM*, 43(7):84–93, 2000.

[10] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan, and Radomír Měch. Visual models of plant development. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 535–597. Springer, 1997.

[11] Przemyslaw Prusinkiewicz and Aristides Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.

[12] Grzegorz Rozenberg. Selective substitution grammars (Towards a framework for rewriting systems). Part 1: Definitions and examples. *Elektrische Informationsverarbeitung und Kybernetik*, 13(9):455–463, 1977.

[13] Grzegorz Rozenberg and Arto Salomaa. *The Mathematical Theory of L Systems*. Academic Press, 1980.

Towards a more flexible Model-Driven Engineering

Juan de Lara[1]

Computer Science Department,
Universidad Autónoma de Madrid,
Madrid, Spain
Juan.deLara@uam.es

Abstract

Model-Driven Engineering (MDE) proposes an active use of models in the different phases of software development. In this approach, models are used to specify, test, simulate, analyse and generate code for the final application. Hence, an MDE process necessitates from the definition of different kinds of model manipulations, like transforming models between different languages, in-place transformations, or code generation.

While it is possible to describe models using general purpose modelling languages, like the UML, the full potential of MDE is unfolded by the use of Domain Specific Modelling Languages (DSMLs), tailored to particular domains. In this way, MDE can be seen as a reutilization approach, because DSMLs and their associated artefacts (transformations, code generators) can be reused across different projects within the domain. However, MDE artefacts are tied to specific DSMLs, and cannot be reused for other similar DSMLs sharing essential characteristics. The end result is that similar transformations have to be implemented repeatedly, even for DSMLs with small variations only.

In this talk, we will discuss several techniques enabling a more flexible use of MDE artefacts, enabling their reuse across different DSMLs. On the one hand, reusability can be obtained by the definition of artefacts over so called *concepts*, gathering the requirements expected from DSMLs for the artefact to be applicable on their instance models. On the other, families of DSMLs can be defined using multi-level modelling, so that artefacts can be defined and applied to sets of related DSMLs. These ideas will be illustrated with the definition of a catalogue of reusable model abstractions.

1 Introduction

Model-Driven Engineering (MDE) aims at increasing the levels of productivity and quality by an active use of models as the main assets in software projects [15]. Hence, models are not used just as passive documentation, but to specify, simulate, analyse and generate code for the final application. Models have a high level of abstraction, contain less accidental details and are more intentional than code. While it is possible to use the UML [13] as a modelling language, it is common to define domain-specific modelling languages (DSMLs) for specific areas of interest [9]. Hence, DSMLs capture the main primitives and abstractions of the domain, which enables gathering the experience and best practices in specific application areas. In this way, engineers work with concepts of the domain (and not of the solution space), reducing the cognitive distance between the problem and its solution. By means of model transformations and code generators, tailored to the specific DSML and application, a large part, or all the application code can be automatically generated.

In MDE, DSMLs are defined through a model describing the main concepts of the domain: its abstract syntax. Such model is called "meta-model" and is normally built using languages

like UML class diagrams, or the MOF [11]. Meta-models are associated a (visual or textual) concrete syntax, specifying how models should be represented. MDE processes also need from model manipulations, like transformations of models between different languages, in-place model transformations (like e.g., optimizations, refactorings or model simulation/animation) or code generations. For this purpose, it is common to use of domain-specific languages, specially tailored for such tasks, like the Atlas Transformation Language (ATL) for model transformations [8], MOF Model-to-text for writing code generators [12], or the Epsilon languages as an integrated set of model-management languages [6].

MDE can be seen as a reutilization approach, because DSMLs and their associated artefacts (transformations, code generators) can be reused across different projects within the domain. However, it is also true that MDE is *type-centric* [4], because the different supporting artefacts (transformations, code generators) are defined over the types of a specific meta-model and cannot be reused for other meta-models, even if they share *essential* structural features. This fact hampers the adoption of MDE in industry because similar transformations have to be repeatedly developed, even for meta-models with only slight differences.

In this talk, we will discuss different alternatives to define more flexible MDE artefacts, which can be reused for different meta-models. Taking ideas from generic programming, transformations can be defined over so-called *concepts* [4, 7], instead of over concrete meta-models. A concept specifies the structural requirements that meta-models need to fulfil for model operations (e.g. transformations) to be applicable on their instances. *Concepts* resemble meta-models, but their elements (classes, references, fields) are variables that need to be *bound* to concrete meta-model elements. Hence, similar to generic programming templates, a *transformation template* is instantiated for a specific meta-model via a *binding*. A binding induces a retyping of the transformation, which then becomes applicable to instances of the particular meta-model. This approach results in reusable transformations, because the concept can be bound to several meta-models, and the transformation can be used with all of them, as shown to the left of Figure 1.

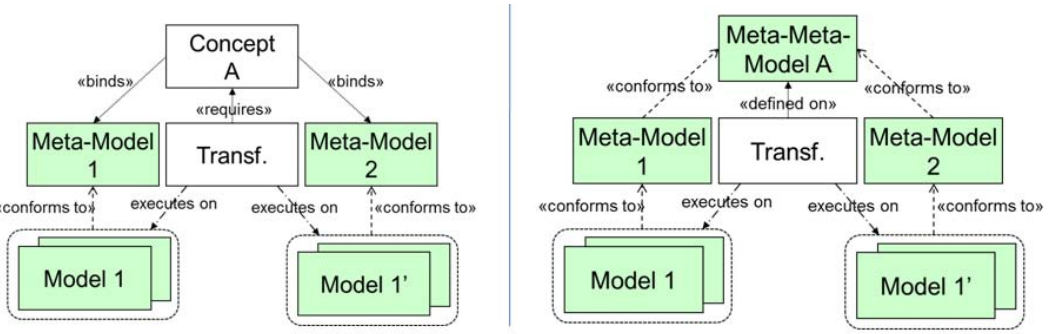


Figure 1: Concept-based genericity (left). Multi-level based genericity (right).

While *concepts* allow a certain degree of reuse, they express structural requirements, which may have been implemented in different ways in different meta-models. For example, a meta-model for Petri nets may represent the number of tokens as an integer attribute or as a class *Token* and a reference from *Place*. We propose two solutions to alleviate this rigidity. First, *hybrid concepts* [4] demand the implementation of some operations (e.g. *getTokens()*) in addition to the binding. This way, hybrid concepts hide behind such operations the structure that may be

implemented differently in different meta-models. Model operations, like transformations, use these operations to implement the functionality. Secondly, we can also use *binding adapters* [14], which are expressions resolving the heterogeneities between a concept and a concrete meta-model. Adapters induce an adaptation of the transformation (via a high-order transformation), which then becomes applicable to the particular meta-model.

Another approach to genericity is the use of multi-level meta-modelling [1, 2, 10]. This approach allows working with an arbitrary number of meta-models, and not just two (i.e. model/meta-model), and permits Domain-Specific Meta-Modelling [3]. This way, it is possible to define meta-modelling languages for particular domains (e.g., process modelling), which can be used to define DSMLs for different applications within a domain (e.g., software process modelling, educational process models, etc). This approach promotes reusability, because transformations can be defined over the top-level meta-model, and then reused for each DSML within the domain, as shown to the right of Figure 1.

As an illustration of the previous techniques, we will discuss the definition of reusable model abstractions for DSMLs [5]

References

[1] Colin Atkinson and Thomas Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[2] Juan de Lara and Esther Guerra. Deep meta-modelling with MetaDepth. In *TOOLS'10*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010. See also `http://astreo.ii.uam.es/~jlara/metaDepth`.

[3] Juan de Lara and Esther Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In *ECMFA12*, volume 7349 of *LNCS*, pages 259–274. Springer, 2012.

[4] Juan de Lara and Esther Guerra. From types to type requirements: Genericity for model-driven engineering. *Software and Systems Modeling*, page in press, 2012.

[5] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. Abstracting modelling languages: A reutilization approach. In *CAiSE'12*, volume 7328 of *LNCS*, pages 127–143. Springer, 2012.

[6] Epsilon. `http://www.eclipse.org/epsilon/`, 2012.

[7] Ronald García, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. *SIGPLAN*, 38(11):115–134, 2003.

[8] Frederic Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. See also `http://www.emn.fr/z-info/atlanmod/index.php/Main_Page`. Last accessed: Nov. 2010.

[9] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE CS, 2008.

[10] Thomas Kühne and Daniel Schreiber. Can programming be liberated from the two-level style?: multi-level programming with deepJava. In *OOPSLA*, pages 229–244. ACM, 2007.

[11] OMG. MOF 2.4.1 specification. `http://www.omg.org/spec/MOF/2.4.1/`.

[12] OMG. MOFM2T 1.0 specification. `http://www.omg.org/spec/MOFM2T/1.0/`.

[13] OMG. UML 2.4.1 specification. `http://www.omg.org/spec/UML/2.4.1/`.

[14] Jesús Sánchez-Cuadrado, Esther Guerra, and Juan de Lara. Generic model transformations: Write once, reuse everywhere. In *ICMT'11*, volume 6707 of *LNCS*, pages 62–77. Springer, 2011.

[15] Markus Völter and Thomas Stahl. *Model-driven software development*. Wiley, 2006.

Record Based Differential Privacy Budget

Hamid Ebadi

Chalmers University of Technology, Göteborg, Sweden
hamide@student.chalmers.se

Abstract

In this paper we propose a new method of privacy budgeting for privacy sensitive databases.

1 Introduction

Organizations willing to release information about their customers to third parties. People are also willing to be influential in election or other kinds of statistical analysis, but having some level of privacy concerning their personal information is more important for them. Differential privacy is a robust standard for data privacy proposed by Dwork based on the idea that the result of analysis should be formally indistinguishable with or without any one record[1].

Differential privacy protects individuals by providing noisy response to statistical queries, so the result of running any arbitrary analysis on two similar databases (symmetric difference close to one) becomes indistinguishable. To achieve this privacy guarantee, access to private data is limited to sanitation functions (M), which are randomized algorithms to add proper amount of noise to the result of computation.

Mechanism M preserve ϵ-differential privacy, if the probability of any specific answer does not significantly change by existence of any one individual record in the data set.

Different platforms for privacy preserving data analysis have already been introduced, like PINQ [3] (Privacy Integrated Queries), Fuzz [2], Airavat [5] and GUPT [4].

Privacy budgeting, which is common in all these platforms is responsible to limit information disclosure. It carefully tracks queries in order to ensure not only an individual query but also its aggregation with previous queries will not violate individual's privacy.

This privacy standard forbids adversary to use the database once the defined privacy budget for database is consumed. Once it happens, database becomes inaccessible even if only a small part of it is involved in queries.

If privacy budget is ϵ, we can independently answer k queries, where the accuracy parameter for the i^{th} query is ϵ_i, and $\sum_i \epsilon_i \leq \epsilon$ without worrying that the aggregation of these k queries violates ϵ-differential privacy. If queries are applied to disjoint subset of data, parallel composition can be used. In these scenarios, instead of summing all privacy guarantees, maximum privacy guarantees of all analyses can be used.

2 Problems With Existing Approaches

Privacy budget for current implementations (PINQ, Fuzz, Airavat and GUPT) is defined for a set of records. For each query an accuracy parameter is defined that decreases the privacy balance for the whole dataset even if it involves only a small subset of records. Even though parallel composition improves database utilization, it is not always applicable. To have a good result in saving privacy budget, it is necessary to find possible ways of executing queries in parallel. Even if the best possible way to run queries in parallel can be easily found, it requires

all queries to be present in the system at the same time. So no dependency between queries are allowed. Another issue with databases that use classic differential privacy is its short lifetime. In the privacy context this means, once we fully analyzed the data for this year (like census database), we cannot promise the same privacy guarantee for the next years, when sensible number of fresh records will be inserted, updated and deleted, but there are still common records between these two databases. This is more sensible when we are interested to run statistical queries on actively changing databases like a list that temporally holds a stream of data. For example a network buffer that its old data is periodically replaced with new data.

3 Record Based Privacy Budgeting

We propose to assign a privacy budget for every single record instead of the whole database. The privacy balance increases when the record is actively participate in an analysis. ($Balance_i = Balance_i + QueryEpsilon$). Whenever privacy budget for a record is reached ($Balance_i > Budget_i$), the system stops disclosing information about that record by ignoring the record and not including it in further analysis. In what follows we briefly discuss the pros and cons of new approach in perspective with existing approach.

1. Maximum utility: Data stored in databases is the most valuable part of IT systems. Collecting this information is time consuming, complicated, and an expensive task that is often subject to restrictions and strict regulations. Therefore it is extremely important to extract as much information as possible from collected data. Using this method more information can be extracted from database as this new system can answer sequence of queries that is not possible to be answered in the classic model.

2. Flexible privacy sensitivity: Since a privacy balance and a budget are assigned to each record, it is possible to have records with different privacy sensitivity. One person can choose to disclose more information about himself without endangering other.

3. Accuracy: PINQuin can accurately respond to a sequence of queries as long as the remained privacy budget for all records involved in aggregation query are more than privacy cost of the query. Running the same queries in PINQuin and PINQ leads to the same results until PINQ stops because its budget runs out ($Balance_i > Budget_i$). At this point PINQ stops responding to new queries while PINQuin can respond to more queries. Records that run out of budget cannot participate in any more analysis. This results in inaccurate analysis if the analysis is missing some records.

4. Dynamic database: This approach helps us have more flexibility and dynamic databases where data can be removed, added or edited without worrying about its privacy impacts. When a new record is added a fresh budget is assigned to it. It is also possible to edit each record but its corresponding budget should remain untouched. Also removing a record before it consumes its budget will not endanger its privacy.

5. Protection against budget side channel: Our model eliminates the privacy budget side channel attack that is shown in "Differential privacy under fire"[2]. In our model all queries will be answered and no query will be blocked, but the accuracy of analysis decreases as more and more records run out of privacy budget.

6. Flexible queries and simplicity: It is suggested to use complex parallel composition to consume less privacy budget. To get best result from parallel composition, all queries

should be present in the system at the same time, which also conveys that the queries should not depend on the result of each other. Our model doesn't enforce any of these restrictions, as a result this makes queries simpler and more flexible.

7. Storage: For every record a data field is required to store privacy balance. This value is updated for those records that are involved in aggregation. If we have records with different sensitivity another data field for storing privacy budget is also needed.

8. Time: Time needed to answer each query in PINQuin is more than PINQ. As in each transformation operation, the system keeps track of involved records and when it turns to aggregation operation, related balance should be updated and it should be decided if that record can participate in analysis or not.

PINQuin is the platform we are designing to demonstrate record based differential privacy. PINQuin follows the same principle and path as PINQ [1], so with small modification in application, PINQ can be replaced by PINQuin. In PINQuin, for each record a privacy budget and a privacy balance are assigned. PINQuin tracks how a record flows between transformations. When an aggregation operation is executed, the privacy balance for every involved record increases. Once the privacy balance for a record reaches its privacy budget, that specific record cannot be used in any other analysis.

As mentioned before accuracy of result decreases as records run out of privacy budget. This method is efficient if queries don't inherently consume all records budget or the number of records that run out of budget are not that high to considerably change the result of analysis. If we have a queue with flow rate R, length L, and each record has budget of B, we can execute ϵ-differentially private queries with the following rate: $\frac{B*R}{L*\epsilon}$. In such scenarios, the buffer is constantly refreshed with new records, so possible run out of budget records is replaced by fresh ones. This kind of analysis on network buffer can be used to have a real-time information about network traffic status without violating privacy of individual record.

We are working on a prototype to demonstrate *"Record Based Differential Privacy"* and to compare it with other differential privacy tools.

References

[1] Cynthia Dwork. A firm foundation for private data analysis. *Commun. ACM*, 54(1):86–95, January 2011.

[2] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. Differential privacy under fire. SEC'11, pages 33–33, Berkeley, CA, USA, 2011. USENIX Association.

[3] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. SIGMOD '09, pages 19–30, New York, NY, USA, 2009. ACM.

[4] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. Gupt: privacy preserving data analysis made easy. SIGMOD '12, pages 349–360, New York, NY, USA, 2012. ACM.

[5] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: security and privacy for mapreduce. NSDI'10, pages 20–20, Berkeley, CA, USA, 2010. USENIX Association.

[1] Privacy Integrated Queries (PINQ), http://research.microsoft.com/en-us/projects/pinq/

Meta-model Evolution with Model Migration based on Graph Transformation*

Florian Mantz
Bergen University College
fma@hib.no

Gabriele Taentzer
Philipps-Universität Marburg
taentzer@informatik.uni-marburg.de

Model-driven engineering [5] (MDE) is a software engineering discipline which focuses on models as abstract representations of domain knowledge rather than on coding. Software is (partly) generated by means of model-to-model and model-to-code transformations. In particular domain-specific modeling languages are developed for a specific purpose and hence allow effective modeling at a high abstraction level. Modeling languages may evolve during their life-cycle for various reasons: the domain may evolve, the modeling language should be extended by additional concepts to enable the generation of larger parts of the software or modeling language designers find a way to simplify their language by e.g. applying conventions not being required to be modeled explicitly. Modeling language evolution often implies that existing models defined by an evolving language need to be migrated (see. Fig. 1). Since the manual co-evolution process is tedious and error-prone, different approaches have been developed that (partially) automate this process.

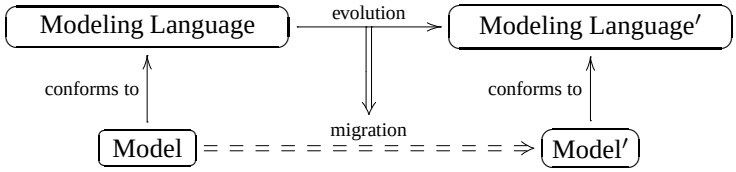


Fig. 1 Model co-evolution: Modeling language evolution and model migration

Modeling languages can be defined using the concept of meta-modeling. Up to now there are several approaches to this co-evolution problem for modeling languages e.g. [2, 7, 10]. Current approaches tackle the challenge of meta-model evolution with model migration basically in four steps (see Fig. 2):

1. Meta-models are evolved by a set of operations, each already related to a migration operation, as done in refactoring. Alternatively, a difference meta-model is calculated and analyzed to reconstruct the sequence of evolution changes.
2. Changes are matched to model migration operations.
3. Migration operations are adapted or added since there is e.g. no migration operation defined for a specific change or another variant of migration is desired.
4. Models are migrated as automatically as possible.

In our research we introduce a new step before model migration: We want to check the migration operations for correctness criteria we have identified. And we want to generate migration operations from evolution operations as far as possible. To be able to consider correctness criteria we

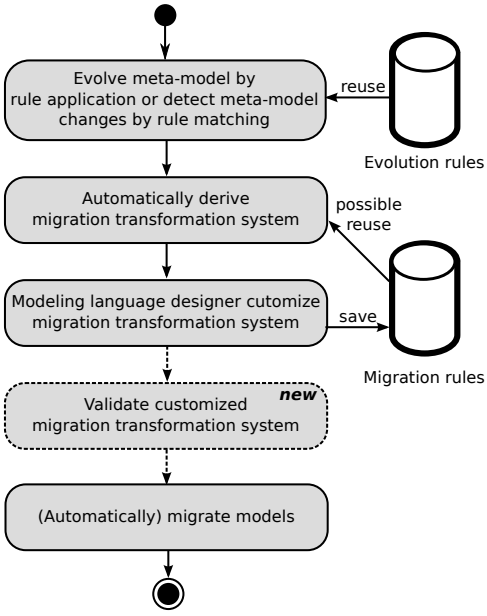


Fig. 2 Meta-model co-evolution process

*This work was partially funded by NFR project 194521 (FORMGRID)

consequently need to consider modeling languages and models on an formal level. We use the formalization of models by graphs that are related to other graphs by morphisms, so that we can apply category theory [1] and the theory of algebraic graph transformation [3]. Formalizing models as graphs is common and fits naturally to the concept of meta-modeling where a meta-graph serves as language definition that restricts other graphs via a graph homomorphism known as typing. Algebraic graph transformation uses two pushouts (see Fig. 3), a construction known from category theory, to rewrite graphs by rules which can be either spans or cospans [4]. We currently work with cospan rules because they allow better synchronization of deletion and creation actions than the usual DPO approach, since the intermediate graphs contain both elements to be deleted and those to be added. Our idea is to consider meta-model evolution and model migration as two graph transformations related by typing morphism (see Fig. 4). The upper transformation in Fig. 4 $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ (with rule $tp := TL \xrightarrow{tl} TI \xleftarrow{tr} TR$ and match tm) describe the meta-model evolution transformation of graph TG while the lower transformation $t : G \overset{p,m}{\Longrightarrow} H$ describe the model-migration transformation of graph G. Properties we have identified to be important for model co-evolution are:

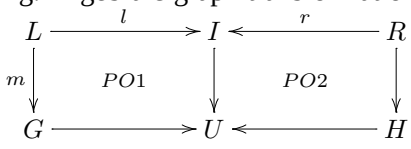


Fig. 3 Graph transformation with a co-span rule on Graph G by match m

1. A model migration result is well-typed i.e. there is a valid typing morphism between the evolved meta-model and migrated models (TH and H in Fig. 4).
2. A predictable migration result is desired. Therefore, it should be clear when model migrations produce a unique result.
3. Evolution and migration rules define reusable transformation knowledge. Reusability should therefore be supported.

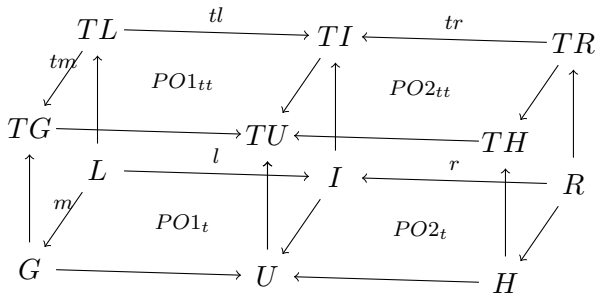


Fig. 4 Co-transformation

4. Often evolved meta-models include several changes being described by transformation sequences. Therefore evolution and migration scenarios should be composable.
5. A model can often be migrated following various variants. This means that a model migration mechanisms should be adaptable.
6. Constraints on meta-models cannot be specified by typing morphisms. Additional meta-model constraints are often given in a suitable constraint language such that the set of valid instances is further restricted. A meta-model evolution approach should be able to deal with such additional constraints in a reasonable way.

In the following we discuss our contributions to each item. Item 1 and 2: in a (weak) adhesive High-Level-Replacement (HLR) category [8, 3], being e.g. the category of attributed, graphs with inheritance as defined in [6] we have identified sufficient conditions for the existence and uniqueness of co-transformations. Identified properties in co-transformations are *match-completeness* and *deletion-reflection*. Match-completeness is defined by a pullback in the left face of the double cube in Fig. 4. It ensures that all elements possibly effected by a evolution transformation are also considered in the migration transformation. In addition, types that are deleted by an evolution transformation require the deletion of their corresponding instances in related migrations. These co-transformations are called deletion-reflecting. A condition that ensures this property is that the right-back face of the double cube in Fig. 4 is a pullback. For match-complete and deletion-reflecting transformations we can state:

Theorem 1 (existence of match-complete co-transformation)**.** *In any (weak) adhesive HLR category: given a co-span graph transformation* $tt : TG \overset{tp,tm}{\Longrightarrow} TH$ *with a monic match* tm *(describing the evolution*

of type graph TG, see Fig. 4) and a graph G typed by TG with typing morphism $t_G\colon G \to TG$ then there exists a co-span graph transformation $t : G \overset{p,m}{\Longrightarrow} H$ (describing a corresponding model migration) such that:

1. *migration rule p is deletion-reflecting wrt. evolution rule tp*
2. *(tt, t) forms a match-complete co-transformation*

Corollary 1 (unique typing of co-transformation)**.** *Given a deletion-reflecting co-transformation rule (tp, p) with an applicable complete match (tm, m) on a type graph TG with instance graph G: then the migration t is uniquely typed by evolution tt (up to isomorphism), i.e. t_U and t_H are uniquely determined (see Fig. 4).*

Theorem 1 and Corollary 1 are proven in [11] for rules with both rule morphisms being monic, and in [9] for rules where only the right rule morphism has to be monic. A framework for constructing match-complete, deletion-reflecting migration rules that leaves space for variants is presented in [11, 9]. Items 3 and 4 in the list above: There are some results known from algebraic graph transformation that we can build on. Graph transformation rules are reusable and there are also composition concepts. Items 5 and partly 3 are subject of our current research: Given a meta-model evolution, our approach needs model-specific migration rules to guarantee match completeness. To come up with a model independent and reusable specification of model migrations, we intend to derive a migration transformation system from a meta-model evolution rule that ensures properties such as match-completeness. The migration transformation system defines a default migration behavior. In addition, it should be customizable by the user since different variants of migrations may be possible especially concerning the creation of elements of new types. These user refinements of the transformation system have to be consistent with its meta-model evolution and should be checked automatically. Item 6 is future work.

References

[1] M. Barr and C. Wells. *Category Theory for Computing Science (2^{nd} Edition)*. Prentice Hall, 1995.

[2] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *EDOC 2008*, pages 222–231. IEEE Computer Society, 2008.

[3] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, March 2006.

[4] H. Ehrig, F. Hermann, and U. Prange. Cospan DPO Approach: An Alternative for DPO Graph Transformation. *EATCS Bulletin*, 98:139–149, 2009.

[5] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[6] F. Hermann, H. Ehrig, and C. Ermel. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In M. Chechik and M. Wirsing, editors, *FASE 2009*, volume 5503 of *LNCS*, pages 325–339. Springer, 2009.

[7] M. Herrmannsdoerfer, S. Benz, and E. Jürgens. COPE - Automating Coupled Evolution of Metamodels and Models. In S. Drossopoulou, editor, *ECOOP 2009*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[8] S. Lack and P. Sobocinski. Adhesive Categories. In I. Walukiewicz, editor, *FoSSaCS 2004*, volume 2987 of *LNCS*, pages 273–288, 2004.

[9] F. Mantz, G. Taentzer, and Y. Lamo. Co-Transformation of Type and Instance Graphs Supporting Merging of Types with Retyping: Long Version. Technical report, Department of Mathematics and Computer Science, University of Marburg, Germany, September 2012. `www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk`.

[10] L. Rose, D. Kolovos, R. F. Paige, and F. A. C. Polack. Model Migration with Epsilon Flock. In L. Tratt and M. Gogolla, editors, *ICMT 2010*, volume 6142 of *LNCS*, pages 184–198. Springer, 2010.

[11] G. Taentzer, F. Mantz, and Y. Lamo. Co-Transformation of Graphs and Type Graphs With Application to Model Co-Evolution. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT 2012*, volume 7562 of *LNCS*, pages 326–340. Springer, 2012.

Code Generation for Protocols from CPN models Annotated with Pragmatics

Kent Inge Fagerland Simonsen[1,2], Lars Michael Kristensen[1] and Ekkart Kindler[2]

[1] Department of Computer Engineering, Bergen University College, Norway
Email: {lmkr,kifs}@hib.no
[2] DTU Informatics, Technical University of Denmark, Denmark
Email: {kisi,eki}@imm.dtu.dk

Introduction. Model-driven software engineering (MDE) [3] provides a foundation for automatically generating software based on models. Models allow software designs to be specified focusing on the problem domain and abstracting from the details of underlying implementation platforms. When applied in the context of formal modelling languages, MDE further has the advantage that models are amenable to model checking [1] which allows key behavioural properties of the software design to be verified. The combination of formally verified models and automated code generation contributes to a high degree of assurance that the resulting software implementation is correct according to the verified properties.

Coloured Petri Nets (CPNs) [2] have been widely used to model and verify protocol software [4], but limited work exists on using CPN models of protocol software as a basis for automated code generation. In this paper we present an approach for generating protocol software from a restricted class of CPN models. The class of CPN models considered aims at being *descriptive* in that the models are intended to be helpful in understanding and conveying the operation of the protocol while at the same time being close to a verifiable version of the same model and sufficiently detailed to serve as a basis for automated code generation when annotated with *code generation pragmatics*. The purpose of the pragmatics is to address the problem that models with enough details for generating code from them tend to be verbose and cluttered.

Our code generation approach consists of two main steps starting from a CPN model annotated with pragmatics: the first step is to construct an *abstract template tree* (ATT). The ATT then directs the code generation in the second step in that each node of the ATT has associated code templates that are invoked to generate code. In the following we first introduce pragmatics and the class of CPN models considered using a small example of a unidirectional data framing protocol. Next, we use the framing protocol example to illustrate the two main steps of the code generation.

CPN model structure and pragmatics. A protocol system consists of a set of *principals* communicating over *channels*. This structure is reflected in the modular structure of the class of CPN models considered: there is a module for each principal and each channel, and our generation approach generates the code for each of the principals. The unidirectional data framing protocol used as an example in this paper consists of a sender principal and a receiver principal communicating over a FIFO channel. Each principal, in turn contains sub-modules for each of the service primitives (methods) that the principal provides to upper layer protocols or applications. To enable code generation, we introduce the concept of *pragmatics* which are syntactical annotations that can be associated with CPN model elements (e.g., places, transitions, and inscriptions) and are indicated between << >>. The pragmatics are either added *explicitly* by the modeller or are *implicit* in that they are inferred automatically derived from the structure of the CPN model. The pragmatics used in this paper fall into two categories: operation pragmatics and control flow pragmatics.

Figure 1 shows the CPN module representing the send method primitive provided by the sender in the framing protocol. Due to space limitations, we only sketch how code generation is performed at the method level and omit the principal and API levels (which are comparably simpler). The transition Send (top) annotated with an <<external>> pragmatic represent the entry point of the send method. The msg parameter specifies that the send method takes a message (msg) as parameter. Using CPNs for modelling protocols typically follows some general principles which clearly separate between control flow elements and data elements. Automatically identifying the control flow elements, however, is sometimes a bit tricky. Therefore, our approach assumes that the control flow pragmatics <<ID>> provides this information, which the modeller had in mind when designing the CPN model.

In the CPN model from Fig. 1, each incoming message is first partitioned (transition Partition) into a set of Outgoing Messages (fragments). Next, the control flow enters a loop (place Start) where each of the fragments are sent together with a flag indicating whether or not it is the last frame of the message. The loop terminates (transition AllSent) after the last frame has been sent and the send method returns. The latter is modelled by the Completed transition annotated with the <<return>> pragmatic.

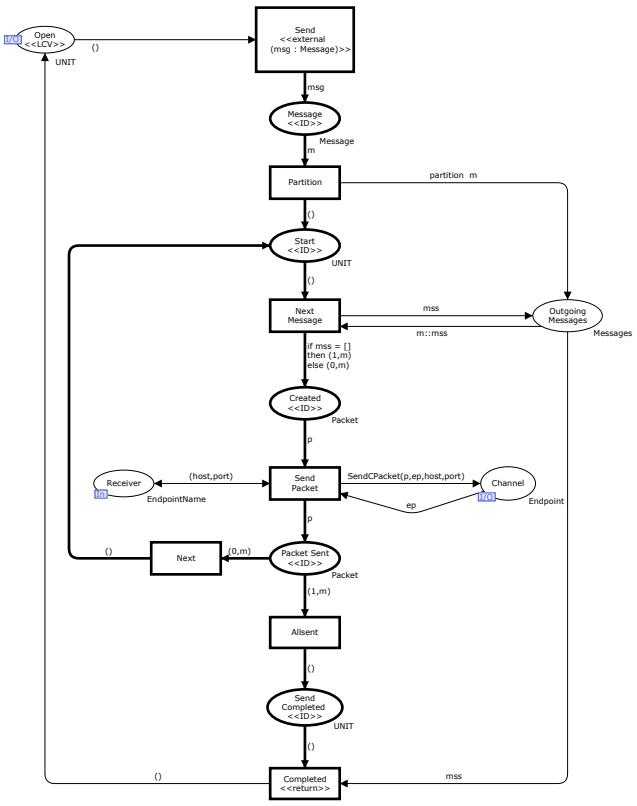


Figure 1: Send CPN submodule (sender principal).

Abstract template tree construction. The abstract syntax tree (ATT) for a CPN module is an ordered tree that has nodes corresponding to explicit and inferred pragmatics. The ATT is derived from the CPN model based on the pragmatics. The ATT represents the control structure of the CPN independently from a specific programming langage. The nodes of the ATT is bound to code templates which are then applied during code generation. When constructing the ATT representing the part of the protocol shown in Fig. 1, the control flow path as determined by the <<ID>> pragmatic is traversed starting from the transition annotated with the <<external>> pragmatic. The <<external>> pragmatic is translated into a *container node* in the ATT which will contain all nodes for this module.

The implicit pragmatics are computed in a preprocessing stage prior to the actual ATT construction. The send module has some implicit pragmatics that are inferred during the traversal of the CPN module corresponding to operations that are to executed at different points along the control flow path of the principal. One example is <<partition>> which is determined by the `partition` function used in the arc expression on the arc from transition Partition to Outgoing Messages. Another example is a <<send>> pragmatic inferred from the model pattern used at the SendPacket transition for accessing the place Channel for sending a packet.

While traversing the CPN module for inferring derived pragmatics, the *block* structure of the CPN model is computed. This block structure characterises the control flow structures of the CPN, such as *branches*, *loops*, and *sequences*. This decomposition is, on the on hand side, used for inferring implict control flow pragmatics – e. g. tagging places with start and end loop pragmatics, where the end loop pragmatics holds a boolean condition for continuing the loop. On the other hand side, the block structure is represented as the ATT, which captures the control flow independently from any specific programming language or platform.

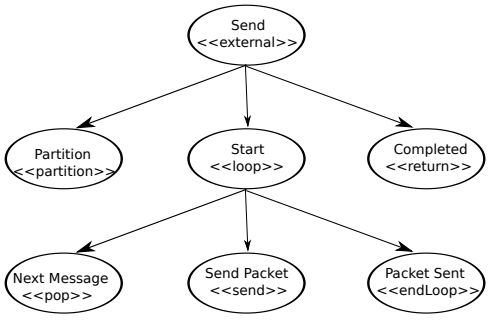


Figure 2: ATT for the send CPN submodule.

Figure 2 shows a simplified representation of the the ATT corresponding to the send CPN module in Fig. 1 (right). The <<external>> root node of the ATT contains nodes for partitioning the message, executing

a loop, and returning. It should be noted that sequences are not explicitly represented in the ATTs but given by the order of child nodes. The <<loop>> node contains nodes for extracting the next fragment, sending a packet, and ending the loop.

Code Generation. After the ATT has been created, code is generated by applying templates for each pragmatic on each node in the ATT. Below we illustrate how code can be generated implementing the framing protocol in the Groovy programming language. In this context, template texts are combined replacing the **%%yield%%** directive in container templates with the code generated for its child nodes.

As an example of a container template, the template for a Loop pragmatic for the Groovy language is given in Listing 1. The template creates a while-loop which continues while the __LOOP_VAR__ variable is true. The body of the Loop is populated by replacing the **%%yield%%** directive with the code generated by the templates of the sub-nodes in the ATT. The __LOOP_VAR__ is updated at the end of the loop by the <<endLoop>> pragmatic which is always present as the last child element of a loop. The <<send>> pragmatic is an example of an operation and is used to send a message over a channel. Listing 2 show the template for the Send pragmatic which requires two parameters: one is the name of the socket that the message should be sent on, and the other is the variable that holds the message to be sent.

Listing 3 shows the generated Groovy code corresponding to the loop in the send method. The code generated for the send pragmatic in the example above is highlighted in Listing 3.

Listing 1: Template for loops in Groovy.

```
%%VARS:__LOOP_VAR__%%
__LOOP_VAR__ = true
while(__LOOP_VAR__){
        %%yield%%
}
```

Listing 2: Template for sending a message.

```
${params[0]}.getOutputStream()
        .newObjectOutputStream()
        .writeObject(${params[1]})
%%VARS:${params[1]}%%
```

Listing 3: Generated code for the loop.

```
__LOOP_VAR__ = true
while(__LOOP_VAR__){
  def  m = OutgoingMessage.remove(0)
  if(OutgoingMessage.size() > 0){
    __TOKEN__ = [1,m]
  } else {
    __TOKEN__ = [0,m]
  }
  Receiver.getOutputStream()
          .newObjectOutputStream()
          .writeObject(__TOKEN__)
  __TOKEN__
  __LOOP_VAR__ = 1 == __TOKEN__[0]
}
```

Conclusion. The code generation approach presented in this paper is still work in progress. Prototypes have been implemented that have allowed initial application of the approach on the data framing protocol and a security protocol. Next steps in the development of the approach will be a formalisation of the concepts and application to larger examples of protocols. A key aspect of this is also the mapping from operations in the CPN models (which are written in Standard ML) into pragmatics and operations on the underlying platform. Here we proposes that operations and corresponding pragmatics can be grouped into packages according to their function in a manner similar to libraries in conventional programming languages. We propose to predefine a set of essential packages with operations that are common to most protocols. However, we acknowledge that it will not be possible to create a full set of operations for the protocol software domain. Therefore, we also provide a way for the modeller to define protocol specific operations. This provides our approach with a high degree of flexibility.

References

[1] C. Baier and J-P Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] K. Jensen and L.M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[3] S. Kent. Model Driven Engineering. In *Proc. of Integrated Formal Methods*, volume 2335 of *LNCS*, pages 286–298. Springer, 2002.

[4] L.M. Kristensen and K. Simonsen. Applications of Coloured Petri Nets for Functional Validation of Protocol Designs. In *Proc. of Advanced Course on Petri Nets*, LNCS. Springer, 2012. To appear.

Distributed Algorithms for Controller Synthesis

Alexandre David, Kim G. Larsen and Zhengkui Zhang

Department of Computer Science, Aalborg University, Denmark
{adavid,kgl,zhzhang}@cs.aau.dk

Abstract

We present a new on-the-fly distributed algorithm for solving timed games based on our previous work on Uppaal-tiga. The originality of the algorithm lies in keeping some locality between states to improve the performance of our two-phase algorithm - forward and backward propagation. Contrary to other works where mostly reachability has been distributed, the same principles underlying our algorithm can be used to define a full TCTL model-checker.

1 Introduction

Uppaal-tiga is a timed game solver for solving controller synthesis problems based on the original algorithm of [7]. The focus of the tool has been so far to improve its implementation and to expand its application range to control with partial observability [6], testing [8, 9], or simulation [4]. Meanwhile computer architecture has been developing toward multi-core and distributed architectures. The future of computing will be distributed as shown by tera-scale computing initiative from Intel [10] having 80 cores working with a message passing architecture.

Controller synthesis and model checking are related formal verification methods, while the former is computationally harder than the latter. Until now many model-checkers have already started to go distributed such as DiVinE [1], LTSmin [3], D-Uppaal [2], or Uppaal-smc [5]. Nevertheless there are still no *distributed synthesis algorithms* exist. Our research aims to fill in the gap by developing a distributed algorithm and a tool. The algorithm can also be generalized to do full TCTL model-checking.

In this paper we present the principle of our distributed algorithm where the main difficulty lies in keeping performance while back-propagating because the algorithm needs information local to every state (immediate successors and predecessors). Although Uppaal-tiga performs well, in practice it is easy to have models that the tool cannot handle due to the size of the model and the ensuing state-space explosion so it is a good candidate for distributing. This would extend the range of models that can handled thanks to increased amount of memory and computing power.

2 Design

One important issue in designing the distributed algorithm is to keep neighbouring states in the state space local to a computing node. The reason behind is that in the backward phase the algorithm will harvest the winning information on all successor states before calculating the winning status of a parent state. Without caching, extra communication will happen for fetching the winning information from remote nodes, and performance decreases.

Figure 1 shows the simplified distributed reachability algorithm. The computation starts as forward search in the master computing node, which is assigned the initial state S_0. The successor states S' branching out from this initial state are generated on this node and a state space partitioning algorithm $h(S')$ is performed to decide which node owns which successor

Initialization (master node 0):

$Passed_0 \leftarrow \{S_0\}$ **where** $S_0 = \{(l_0, \vec{0})\}^{\nearrow}$;
for $S' = Post_\alpha(S_0)^{\nearrow}$ **do**
  $Waiting_A \leftarrow \{(S_0, \alpha, S', F)\}$ **where** $h(S') = A$;
  $Win[S']_0 \leftarrow \emptyset$;
$Win[S_0]_0 \leftarrow S_0 \cap (\{\texttt{Goal}\} \times R^X_{\geq 0})$;
if $S_0 \in Win[S_0]_0$ **then**
  terminate;

Main (node $l \in [0..\mathrm{N}-1]$**):**

while $\bigcup_{i\in[0..\mathrm{N}-1]} Waiting_i = \emptyset$ **do**
  $e = (S, \alpha, S', D) \leftarrow \mathrm{Pop}(Waiting_l)$;
  if $(D == F) \wedge (S' \notin Passed_l)$ **then** ▶ forward search
    $Passed_l \leftarrow Passed_l \cup \{S'\}$;
    $Win[S']_l \leftarrow S' \cap (\{\texttt{Goal}\} \times R^X_{\geq 0})$;
    for $S'' = \texttt{Post}_\alpha(S')^{\nearrow}$ **do**
      $Waiting_A \leftarrow Waiting_A \cup \{(S', \alpha, S'', F)\}$ **where** $h(S'') = A$;
      $Passed_l \leftarrow Passed_l \cup \{S''\}$;
      $Win[S'']_l \leftarrow \emptyset$ **if** $S'' \notin Win_l$;
    if $Win[S']_l \neq \emptyset$ **then**
      $Win[S']_P \leftarrow Win[S']_l$ **where** $h(S) = P$;
      $Waiting_l \leftarrow Waiting_l \cup \{e\}$;
  elif $(D == F) \wedge (S' \in Passed_l)$ **then** ▶ trigger backward propagation
    $Waiting_P \leftarrow Waiting_P \cup \{(S, \alpha, S', B)\}$ **where** $h(S) = P$;
  else ▶ backward propagation
    $Win^* \leftarrow Pred_t(Win[S]_l \cup \bigcup_{S \xrightarrow{c} T} Pred_c(Win[T]_l),$
      $\bigcup_{S \xrightarrow{u} T} Pred_u(T \setminus Win[T]_l)) \cap S$;
    if $(Win[S]_l \subsetneq Win^*)$ **then**
      $Win[S]_l \leftarrow Win^*$;
      for $g = (S^*, \tau, S)$ **where** $S = Post_\tau(S^*)^{\nearrow}$ **do**
        $Win[S]_P \leftarrow Win[S]_l$ **where** $h(S^*) = P$;
        $Waiting_P \leftarrow Waiting_P \cup (S^*, \tau, S, B)$ **where** $h(S^*) = P$;
  endif
endwhile

Figure 1: Simplified Distributed Reachability Algorithm

state. If a successor state belongs to the same computing node, it is stored locally, otherwise it is sent to its target node by message passing and stored in the remote waiting queue $Waiting_A$. Meanwhile the winning sets of all successor states are created on this node, even if some states are stored remotely. In the latter case, the winning sets are called *cached* winning sets.

Once a remote computing node picks a state from its local waiting queue, it starts to generate the successors of this state, and the partitioning algorithm works in a similar fashion. Gradually all computing nodes start to do forward search in the state space. The procedure can be illustrated by a simple example in figure 2. In computing node 0 three successor states are generated from initial state S_0. One successor state S_1 is stored locally, and the other two (S_2, S_3) are sent (brown solid line) separately to node 1 and 2. The winning sets of all

three successor states are create and two of them are cached ($Win[S_2]_0$, $Win[S_3]_0$). State S_2 is received and used to find successor states on node 1. The same with S_3 on node 2.

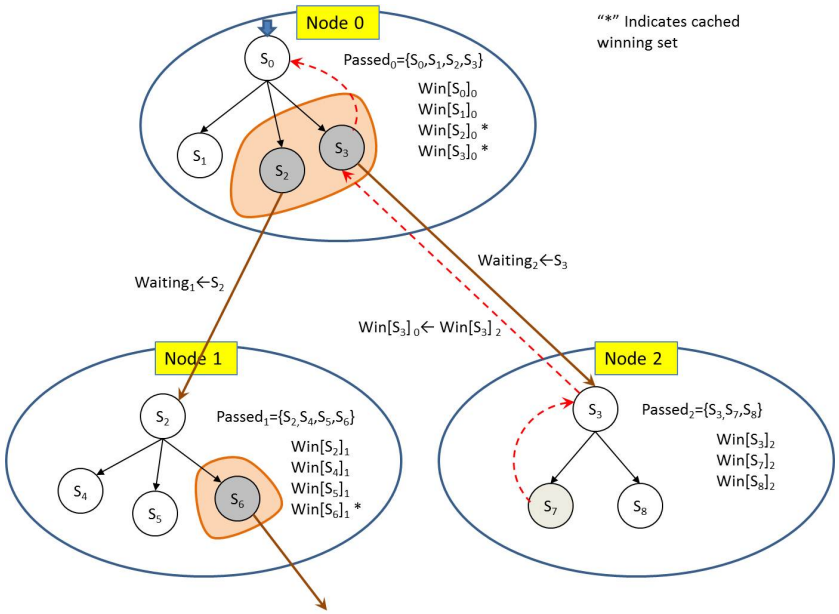


Figure 2: Example Run of a Distributed Reachability Game

When a goal state is reached, its winning set $Win[S']_l$ becomes nonempty and the backward propagation is activated. Because winning sets of remote successor states are cached with their parent state, a parent state can always access to the winning sets or cached winning sets of its successor states. If new winning information is obtained in $Win[S]_l$, the algorithm updates cached winning set of S on all predecessors S^* then backward reevaluates S^*. The computation terminates when there are no states left in the waiting queues among all computing nodes.

Figure 2 also shows a scenario of the backward phase. On node 2, state S_7 is found winning and backward propagation (red dashed curve) starts at its parent state S_3. The cached winning set of S_3 is updated on node 0, and reevalution will finally back-propagate to initial state S_0.

References

[1] J. Barnat, L. Brim, M. Češka, and P. Ročkai. DiVinE: Parallel Distributed Model Checker (Tool paper). In *Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC 2010)*, pages 4–7. IEEE, 2010.

[2] Gerd Behrmann. Distributed reachability analysis in timed automata. *STTT*, 7(1):19–30, 2005.

[3] S. C. C. Blom, J. C. van de Pol, and M. Weber. Ltsmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification, Edinburgh*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359, Berlin, July 2010. Springer Verlag.

[4] Peter Bulychev, Thomas Chatain, Alexandre David, and Kim G. Larsen. Efficient on-the-fly algorithm for checking alternating timed simulation. In *Proceedings of the 7th International Conference on Formal Modeling and Analysis of Timed Systems*, number 5813 in LNCS, pages 73–87. Springer, 2009.

[5] Peter Bulychev, Alexandre David, Kim G. Larsen, Axel Legay, Marius Mikucionis, and Danny Bøgsted Poulsen. Checking & distributing statistical model checking. In Alwyn E. Goodloe and Suzette Person, editors, *4th NASA FORMAL METHODS SYMPOSIUM*, volume 7226 of *LNCS*, pages 449–463. Springer, 2012.

[6] F. Cassez, A. David, K. G. Larsen, D. Lime, and J.-F. Raskin. Timed control with observation based and stuttering invariant strategies. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *LNCS*, pages 192–206. Springer, 2007.

[7] Franck Cassez, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR'05*, volume 3653 of *LNCS*, pages 66–80. Springer–Verlag, August 2005.

[8] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Cooperative testing of uncontrollable real-time systems. In *4th workshop of Model-Based Testing (MBT'08)*, 2008.

[9] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Timed testing under partial observability. In *2nd IEEE International Conference on Software Testing, Verification, and Validation (ICST'09)*, 2009.

[10] Timothy G. Mattson, Rob Van der Wijngaart, and Michael Frumkin. Programming the intel 80-core network-on-a-chip terascale processor. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

Modern Software Development - A Synonym for "Re-inventing the wheel" over and over again

Arne Styve[1]

Offshore Simulator Centre, Ålesund, asty@offsim.no

Even though the software industry seems to have come a long way, it is still in its childhood. Too many programming languages and too many development environments linked to a specific opersting system only cause that we are still "re-inventing the wheel" over and over again.

This talk will focus on how this situation creates unnecessary challenges when designing advanced offshore simulators. Some good initiatives to try to overcome these problems will also be discussed.

Model-Driven Engineering of Maritime Systems

Adrian Rutle[1] and Hans Georg Schaathun[1]

Høgskolen i Ålesund, postboks 1517, 6025 Ålesund
{adru,hasc}@hials.no

1 Introduction

It is well-known that many features are moving from hardware to software, which is thus playing an increasing role in most engineering systems and other applications, at an increasing share of the cost. Many computer scientists have long recognised that software complexity is increasing much more rapidly than the improvements on programming languages and the methods to manage the complexity. Where software, measured in lines of code, is estimated to grow at a factor of ten per decade, very little has happened to programming languages since the introduction of object orientation around 1970.

In this project we are considering challenges and possible solutions within software for the maritime sector, in particular the development of training simulators for advanced marine operations. In the design of such training simulators, we see two major software challenges. Firstly, systems are composite and depend on the collaboration of programmers, electronic engineers, mechanical engineers, mathematicians and others. Each discipline will have to contribute to the code, but common-place languages of the day require a programming specialism to understand. Raising the level of abstraction, programming in a language which is closer to the shared mathematical syntax of the disciplines, we would reduce the risks of miscommunication, ease validation, and make the process less error-prone.

The second challenge is reuse of software components. The maritime industry is very fragmented, and every system will depend on modules from different vendors, each with their own favoured platform. Considering training simulators, we may need a propeller model, a hull model, and an ocean model from three different vendors. With different OS, language and API specification we have a problem. Again, modelling at a higher level of abstraction, a platform independent level, it should be easier to automate code generation for the platform of choice.

Our approach to handle these challenges is based on model-driven engineering (MDE), a development methodology where models are considered the first class artifacts in the development process. In MDE we shift the focus from code – the most important artefact in traditional development processes – to abstract models. These models are used in automated transformations to generate executable code. In this way, we achieve platform and architecture independence, contributing to reusable components that are easier to specify and to compose. In particular, we develop a domain specific modelling language (DSML) which can be used by various domain experts with different expertise to specify various components of the system, then we define connectors between these components based on their structural and behavioural properties.

2 Training Simulator for Marine Operations

A *system* is an object or collection of objects where we want to study the properties [2]. It may be a building, a bridge, an aircraft, a ship, or a piece of software. A *model* of a system is (also) a system, representing the original system. Models may be abstracted or simplified to focus on particular perspectives that are relevant to specific experiments or analyses, thus leaving out details which may be less relevant. Experiments performed on a model representing

the original system called *simulation*, and serve to demonstrate the dynamic behaviour of the system. A simulator model can *describe* an existing ship system, and at the same time *prescribe* a software system to simulate the behaviour of the ship system.

Now we outline a simplified ship simulator model. The model consists of the following main components: Hull, Controller, Thruster Control, Actuator and Propulsor (see Figure 1). Of these components, Hull and Propulsor are models of physical systems, while Actuator is a model of the software used to compensate for the latency in real systems in order to make the simulation more realistic. Controller may be a model of software (e.g., Dynamic Positioning System) or of physical system (e.g., Joystick). The Thruster Control is a model of the software which communicates with Controller and Actuator.

Figure 1 shows the ship simulator model specified in the Eclipse modeling Framework (EMF) [5]. The simulator software can be generated from this model; e.g., at any time, we can query the position of the ship, its speed and direction. In addition, we can perform queries on Actuator and Propulsor to retrieve information about these components' states.

The internal behaviour of the physical components are specified in Modelica [2]. That is, the structural model is a skeleton which can be configured with different behaviour models. The components which communicate with Modelica models, for example, `PropulsorPhysicalModel` and `HullPhysicalModel`, delegate method calls from `Propulsor` and `Hull`, respectively, to the Modelica models representing these physical systems. Compatibility between the Modelica models and the rest of the system is facilitated by a common metamodel for physical units, including the *Système International* (SI) which is already supported by Modelcica as a subset. Some of these types are shown in the lower right hand part of the figure.

Building training simulators that are suitable for training for advanced marine operations requires accurate modelling of the physical components. These components are usually modelled by different vendors. Connectors between these components are defined to represent how a

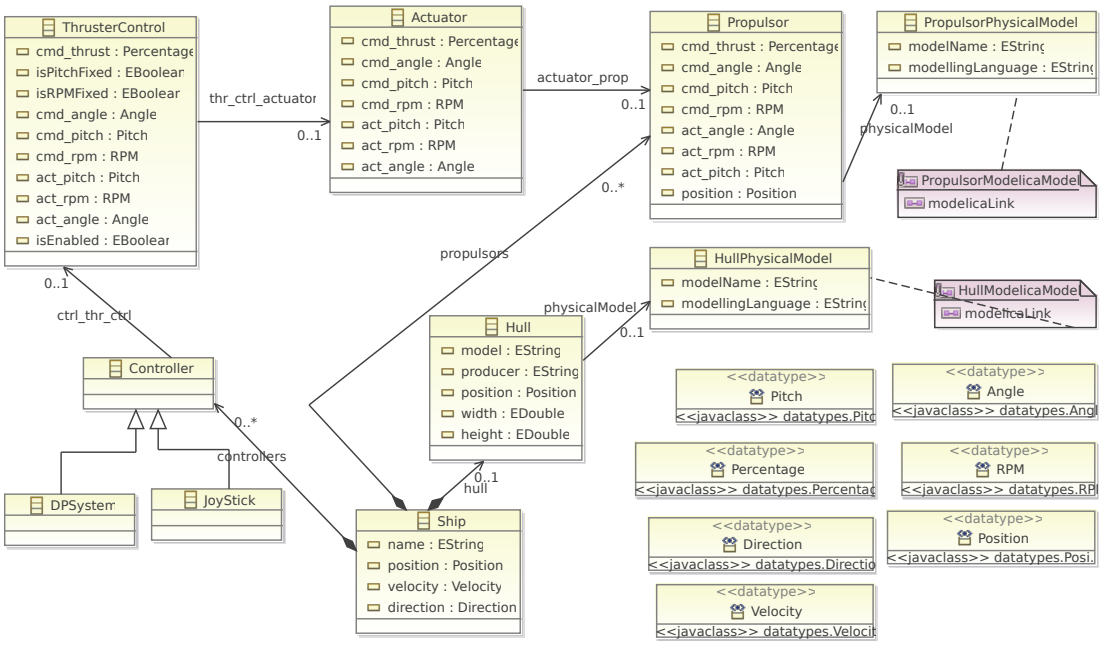


Figure 1: Ship model in EMF

component influences and is influenced by the overall system (or other components). In order to define these connectors, we could connect e.g. propellers and hulls in Modelica. However, this would lead to close coupling of these components making reuse a challenge. This would also require some knowledge in Modelica language and the underlying mathematics for the physical components. Therefore, we define these links at the model level, as in Figure 1. We know in advance how the propellers, engines, gears, etc. are connected to the hull to construct a ship, and use this information to automatically generate code for connecting the Modelica models that represent these components. The end result is that both the structural ship model and the behavioural Modelica models will be compiled to executable code.

3 Related and Future Work

Graphical Modelica Modelling Language (ModelicaML) is a UML profile for Modelica which enables an integrated modelling and simulation of system requirements and design [4]. ModelicaML combines UML and SysML's standardized graphical notations for system modelling with Modelica's strength in modelling and simulation of physical systems. It makes possible to create executable specification and analysis models that can be simulated in discrete and continuous time. SysML4Modelica is another UML profile that is designed to standardize the ralationships between OMG's technologies and Modelica [3]. This profile can be used to annotate SysML elements with Modelica specific information. SysML4Modelica comes with a transformation – SysML-Modelica – that generates Modelica code from SysML design models. These profiles and the transformation SysML-Modelica is all based on the OMG's well-known technologies.

The Modelica community has specified the Functional Mock-up Interface (FMI) [1] to enable model exchange and deployment across tools. This API is C-oriented (as Modelica otherwise,) thus some machine architecture, operating system and language dependence is expected.

Currently we are investigating the usability of the DSML for specification of single systems such as a ship or a crane. In future, the DSML will be extended to enable specification of training scenarios involving a various number of ships, cranes, winches, etc. Thus, we will generalise the approach from building models by connecting different components to building scenarios by connecting various models.

In this article we have outlined a solution for the development of maritime training simulators based on the MDE. The approach facilitates the development and composition of software components on the model level, and later to generate executable code that can be simulated in a training environment. We have specified a model for ship with links to Modelica models describing the behaviour of the physical components.

References

[1] T. Blochwitz et al. The functional mockup interface for tool independent exchange of simulation models. In C. Clauß, editor, *Proc. of the 8th Int'l Modelica Conference*, pages 105–114. Linköping University Electronic Press, March 2011.

[2] P. Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, 2004.

[3] C. Paredis et al. An overview of the sysml-modelica transformation specification. In *2010 INCOSE International Symposium*, July 2010.

[4] W. Schamai, P. Fritzson, C. Paredis, and A. Pop. Towards unified system modeling and simulation with modelicaml: Modeling of executable behavior using graphical notations. In *Proc. of the 7th Int'l Modelica Conference*, pages 612–621. Linköping University Electronic Press, 2009.

[5] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0 (2^{nd} Edition)*. Addison-Wesley Professional, 2008.

Fully Abstract Trace Semantics of Low-level Protection Mechanisms

Marco Patrignani and Dave Clarke

IBBT-DistriNet, Dept. Computer Sciences, Katholieke Universiteit Leuven

Abstract

Fine-grained program counter-based memory access control mechanisms can be used to enhance low-level machine models to become the target of secure (fully abstract) compilation schemes. A secure compilation scheme reduces the power of a low-level attacker with code injection privileges to that of a high-level attacker which generally does not have such privileges. The existing trace semantics for a fine-grained program counter-based memory access control mechanism is not fully abstract, thus the protection mechanism it models cannot be used as the target of a *provably* secure compilation scheme. This paper shows why is such a fully abstract trace semantics needed, and proposes a correction to the existing trace semantics that makes it fully abstract and thus capable of supporting a secure compilation scheme.

Low-level machine code offers virtually no protection mechanism from an attacker that has code injection privileges, who is free to read sensible data and disrupt the execution flow with malicious code. A way to defend against these kind of attacks is by employing a fine-grained program counter-based memory access control mechanisms (FPMAC). The idea behind recent FPMACs implementations [3, 5, 8, 9], is to run sensitive code in isolation, so that malicious low-level code cannot tamper with it. Although details of these works differ, the FPMAC protection mechanism can be summarized as follows. The memory is logically divided into a protected and an unprotected section. Protected memory is further divided into a code and a data section. The code section contains a number of entry points: addresses which unprotected memory instructions can jump to and execute. The data section is accessible only from the protected section. The following table provides a representation of the access control model enforced by the protection mechanism.

From\ To	Protected			Unprotected
	Entry Point	Code	Data	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

This protection mechanism can be the target for a secure (fully abstract) compilation scheme, as the works of Agten *et al.* [2] and Patrignani *et al.* [6] have recently shown. A fully abstract compilation scheme preserves and reflects contextual equivalence at both high- and low-level code, thus it is well suited to expressing the preservation of security policies through compilation. Contextual equivalence is a relation between two programs C_1 and C_2 that cannot be distinguished by a third program called the tester. Formally, for all contexts $\mathbb{C}$ with a hole, if that hole is filled by either C_1 or C_2, the behavior of the context does not vary: $C_1 \simeq C_2 = \forall \mathbb{C}.\mathbb{C}[C_1] \uparrow \iff \mathbb{C}[C_2] \uparrow$, where $\uparrow$ denotes divergence. Contextual equivalence can be used to model security policies as follows: saying that variable f of program C is confidential is equivalent to saying that C is contextually equivalent to a program C' that only differs from C in its value for f. Denote a high-level program C and its compilation $C^{\downarrow}$, a fully abstract compilation scheme is indicated as follows: $C_1 \simeq C_2 \iff C_1^{\downarrow} \simeq C_2^{\downarrow}$ [1].

In the proof of full abstraction for a compilation scheme, the most important direction is completeness, which can be stated as: $C_1^{\downarrow} \not\simeq C_2^{\downarrow} \Rightarrow C_1 \not\simeq C_2$. The completeness result can be proven by devising an algorithm that is capable of creating a high-level context that differentiates high-level component C_1 from C_2, given the low-level context that differentiates their compiled counterparts. Since low-level contexts are simply memory regions filled with instructions and a hole, they do not provide an inductive structure and thus offer little help in a proof. In order to circumvent this problem, the low level language can be equipped with a fully abstract trace semantics, denoted $\mathsf{Trace}_{\mathsf{L}}(M)$. Full abstraction of the trace semantics means that the notions of trace semantics and of contextual equivalence coincide [7]: $\mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2) \iff M_1 \simeq M_2$. This result grants an assumption of the form: $M_1 \not\simeq M_2 \Rightarrow \mathsf{Traces}_{\mathsf{L}}(M_1) \neq \mathsf{Traces}_{\mathsf{L}}(M_2)$, which can be used in the completeness direction of a a proof of full abstraction of a compilation scheme as follows: $\mathsf{Traces}_{\mathsf{L}}(C_1^{\downarrow}) \neq \mathsf{Traces}_{\mathsf{L}}(C_2^{\downarrow}) \Rightarrow C_1 \not\simeq C_2$. At this point, the proof is concluded by devising an algorithm that takes two different low-level traces as input and produces a high-level context that differentiates between two programs, as Agten *et al.* [2] and Patrignani *et al.* [6] have shown.

Flawed Trace Semantics for FPMAC-enhanced Low-level Languages

The first trace semantics for an untyped assembly language that adopts a FPMAC protection mechanism was presented by Agten *et al.* [2], based on labels defined by the grammar:

$$\Lambda ::= \alpha \mid \tau \qquad \alpha ::= \gamma? \mid \gamma! \qquad \gamma ::= \texttt{call}\ a(\overline{v}) \mid \texttt{ret}\ v$$

Those labels capture information flowing from protected to unprotected memory via decoration ! and the other direction via decoration ?. What is not captured by that trace semantics is other information flow, for example when protected code writes in unprotected memory. Consider two low-level programs M_1 and M_2 that exhibit the same trace semantics. If M_1 performs a write in unprotected memory but M_2 does not, an external program is able to tell whether it is interacting with M_1 or M_2 by simply monitoring the changes happening to unprotected memory. This result clearly contradicts assumption $M_1 \not\simeq M_2 \Rightarrow \mathsf{Traces}_{\mathsf{L}}(M_1) \neq \mathsf{Traces}_{\mathsf{L}}(M_2)$, making the trace semantics not fully abstract.

To make the semantics fully abstract, the information exchanged between protected and unprotected memory must be restricted to what appears on the labels of the trace. Information can be exchanged between protected and unprotected memory in three ways: via reads and writes in memory, via values in registers and via flags. The FPMAC protection mechanism prevents external memory to write inside the protected memory, so the only dangerous reads and writes are from the protected memory towards unprotected one. Two possible solutions arise, namely, change the language: either those reads and writes are prohibited, or change the semantics: those reads and writes are captured by the labels in the trace semantics [4]. Values in registers are captured by the actions of the traces, however during a return, only one value is passed, thus all registers besides that one containing the communicated value must be reset to a default value. The same reasoning applies to flags, which must also be reset to a default value when the program counter jumps from protected to unprotected memory. An alternative to registers and values resetting would be to include them in the labels.

Proof Sketch of Full Abstraction of FPMACs Trace Semantics

The proof of full abstraction of the trace semantics, $\mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2) \iff M_1 \simeq M_2$, is split in two cases, one for each direction of the co-implication.

The completeness case: $M_1 \simeq M_2 \Rightarrow \mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2)$ is equivalently stated as: $\mathsf{Traces}_{\mathsf{L}}(M_1) \neq \mathsf{Traces}_{\mathsf{L}}(M_2) \Rightarrow M_1 \not\simeq M_2$. This can be proven in a known fashion: devise an algorithm that takes in input two different low-level traces $\overline{\alpha_1}$ and $\overline{\alpha_2}$ and the two low-level programs M_1 and M_2 generating them and outputs a low-level program M that can interact with M_1 and M_2 such that M diverges only with one of the two low-level programs. The input traces are sequences of actions such that even-numbered actions are messages from M to either M_1 or M_2 and odd-numbered actions are messages from either M_1 or M_2 to M. Since the traces are different by hypothesis, there exists an odd-numbered index j for which $\alpha_1^j \neq \alpha_2^j$. Based on the difference that is found in those odd-numbered actions, the algorithm outputs M so that M is able to terminate in a case and diverge in the other. Even though details of the algorithm are missing, the algorithm sketched here seems feasible to prove the completeness.

Let us now give an intuition behind the proof strategy devised for the soundness case: $\mathsf{Traces}_{\mathsf{L}}(M_1) = \mathsf{Traces}_{\mathsf{L}}(M_2) \Rightarrow \forall \mathbb{C}.\mathbb{C}[M_1]\uparrow \iff \mathbb{C}[M_2]\uparrow$. Coinductively define an equivalence relation between the states of the execution of $\mathbb{C}[M_1]$ and $\mathbb{C}[M_2]$. Break the operational semantics in two sets of transitions, based on whether the program counter points to protected or unprotected memory. Transitions where the program counter moves within unprotected or within protected memory are τ transitions in the trace semantics, thus they preserve state equivalence. Transitions where the program counter jumps from unprotected to protected memory preserve state equivalence as they are generated by $\mathbb{C}$, they are ?-decorated transitions in the trace semantics. Transitions where the program counter jumps from protected to unprotected memory must be proven to preserve state equivalence, they are !-decorated transitions in the trace semantics. This last point can be proven with any of the suggested solutions since they force all communicated information to be on the labels of the traces. Although the proof still needs to be formally carried out, we believe this proof strategy to be sound.

References

[1] Martín Abadi. Protection in programming-language translations. In Jan Vitek and Christian D. Jensen, editors, *Secure Internet programming*, pages 19–34. Springer-Verlag, London, UK, 1999.

[2] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. Secure compilation to modern processors. In *IEEE*, CSF, pages 171 – 185, 2012.

[3] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 375–388, New York, USA, 2011. ACM.

[4] Pierre-Louis Curien. Definability and full abstraction. *Electron. Notes Theor. Comput. Sci.*, 172:301–310, April 2007.

[5] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010.

[6] Marco Patrignani, Dave Clarke, Pieter Agten, and Frank Piessens. Secure Compilation of Object-Oriented Components, October 2012. Submission in preparation.

[7] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[8] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB complexity for security-sensitive applications: three case studies. *SIGOPS Oper. Syst. Rev.*, 40(4):161–174, April 2006.

[9] Raoul Strackx, Frank Piessens, and Bart Preneel. Efficient isolation of trusted subsystems in embedded systems. In *SecureComm*, pages 344–361, 2010.

Dynamic Structural Operational Semantics (preliminary report)

Cristian Prisacariu

Dept. of Informatics, Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
cristi@ifi.uio.no

1 Introduction

We are interested in dynamic aspects of programming languages, with important examples being dynamic software updates for imperative languages like C [5,6], and dynamic class upgrades for object-oriented languages [1,7]. The nature of such dynamic aspects is different than normal programming (control flow) constructs. Yet the semantic interpretation of the dynamic operations is today given using (rather intricate styles of) structural operational semantics (SOS) because this is the formalism of choice for giving semantics to the standard language constructs.

We want to achieve separation of the dynamic constructs from the standard execution constructs, at the level of the SOS specification. To achieve such a separation we introduce *Dynamic SOS*. We are taking a modular approach, following the work of P. Mosses [3], and define dynamic SOS using notions of category theory. To facilitate an easy adoption among the users of SOS (and to follow a goal Plotkin had [4] of "simple mathematics") we use in our presentation terminology close to standard SOS semantics of programming languages.

Our aim with dynamic SOS (DSOS) is to have a general method for giving semantics to runtime update operations, where the works of [1,6] on dynamic software updates will constitute our running examples. Following the work on modular SOS (see also [2]) we represent the data manipulated by the program at runtime, as a category, where the morphisms give the way in which data is manipulated (inspected, changed, or generated). Essentially, configurations in DSOS consist only of the program term (the execution code), where the extra information needed by the program is flowing on transitions which are arrows in the *data category*. These give semantics to the normal program execution constructs. The update operations use a special updated data information in the data category and associate a composition operation which is theoretically given as an endofunctor on the data category. The update constructs are given semantics using these functors and represent *jumps* in the transition system.

2 Dynamic SOS

We consider the syntax of a programming language to be defined as containing *execution operations* and *dynamic update operations*; i.e., program terms are built using functional symbols from $\Sigma_{exe} \cup \Sigma_{upd}$. Normally the signature Σ_{exe} would be multi-sorted and contain constructs like binary sequential or parallel composition, assignments of expressions to variable, function or class declarations, loops, conditionals, and so on. New to these would be the update operations of Σ_{upd}. We will denote a member of Σ_{upd} as upV_i where in this case the letter V is meant to visually indicate the relation of this operation with updating *variable names*, and $i \in Idx$ may be an index from a complex indexing sets; e.g., an indexing can be given by sets of names of variables occurring in the rest of the execution syntax ($Idx = 2^{Var}$). The syntax of the programming language is of lesser interest for the dynamic aspect of SOS. The data manipulated by the program and the update information will be of more interest in the following.

The SOS semantics of a program term is given in terms of a labeled transition system (LTS) of a special kind. An LTS is formed of configurations Γ and labeled transitions between configurations, $\longrightarrow \subseteq \Gamma \times \mathbb{L} \times \Gamma$, where the set of labels $\mathbb{L}$ will be defined shortly.

A *configuration* of a system is formed of a syntactic term t generated by the syntax above (this is the remaining program code to be executed). In standard SOS this is paired with a *data* term (often called the context or the environment) which is the auxiliary information a program needs for running. We prefer to call this term *data*, because it is not of an executable nature, but only of a mutable (read/write) nature, storing information needed by the execution term t for running. In the process algebras community, the notion of context is important in defining notions of congruence. In the distributed systems area, the notion of environment normally refers to the other processes running in parallel with the ones we model. Our results on DSOS may be equally applied to the above areas. Therefore we avoid confusion, and use the term *data*.

In DSOS, the data term is involved in the complex labels, and the configurations are, as in process algebras, formed of only the program term.

The data may have complex structure. The minimal structure required is that of a composition of a mutable *program data* and an *update data*. The program data is supposed to be used by the execution operations, and is usually different forms of heap or type environments, stores, or observable (communication) histories. The update data is supposed to be used by the dynamic update operations in a read-only fashion (relaxations of this, where the update data can also be mutated, are nor investigated here).

$$D \otimes U$$

The $D \otimes U$ is a product of categories, where the composing categories are usually a discrete category if we work with a read-only environment-like component, or is a total category with morphisms between any two objects if we are involving a read/write structure like a store, or in the case of write-only structures like communication histories is a single object category with morphisms forming a monoid. These seam to be all that is needed in practice, but other categories may be envisaged. The objects of such categories are the structures (like heaps, stores, histories) the program works with at all their possible stages. The morphisms of these categories give the way this data is manipulated by the program term. These morphisms are the labels between the configurations of our LTS; i.e., $\mathbb{L} \triangleq Morph(D \otimes U)$.

There is a close correlation between the structure of U and that of D in the sense that for each category component of D that we intend to update at runtime there is a discrete category component of U and the objects of this U component are of the same structure as the objects of the D component, e.g., both have stores. The minimal structure that we impose on a U component is to be a discrete category with a single object. We may complicate this structure in the case of incremental updates. The discreteness of the categories makes U read-only.

The execution of one update operation is associated to performing a composition of the two data structures $D \otimes U$ involving their respective components, to obtain some new $U' \otimes D'$, where normally U' is a substructure of U (where the notion of substructure is classically understood) and D' can be completely different than D. Each composition is captured through an endofunctor on $D \otimes U$. Intuitively, changing D would be done through deletions and additions. Whereas changing U would be done only through deletions; all done by the functor.

The functors are also taken as labels on transitions between the configurations of the LTS. A *computation* in our LTS with these special labels is taken to be a sequence of transitions s.t. the labels are composable in the data category; i.e., for two morphism transitions, possibly separated by one or more functor transitions, $\xrightarrow{\alpha_1} \ldots \xrightarrow{F_i} \ldots \xrightarrow{\alpha_2}$ the origin object of α_2 must

be the same as the target object of α_1 under the application of the intermediate functors in the right order (i.e., $F_n(\ldots F_1(o'_1)) = o_2$ where we denote target objects by prime and origin objects unprimed). In the kind of categories that we mentioned above the functors are defined solely by their application to the objects, where their application to morphisms is deduced from that; i.e., in a discrete category for a morphism α from some object o to itself, the functor $F(o) = w$ gives the morphism $F(\alpha)$ as the unique morphism from $F(w)$ to itself; in a total category (as for mutable stores) a morphism α from o to w is taken to the unique morphism from $F(o)$ to $F(w)$. We are not concerned with write-only components of D since these do not enter under the application of the morphisms.

All the presentation up to now was aimed at dynamic updates as found in the literature [1,6] which update only such data information that the program term uses during execution. But we may want to modify the executing code all together. This is possible if we view the code as data, having one component of D keeping track of the current executing code. This is done using a program counter variable which is updated by all execution operations. This is kept together with the actual execution code term t, where the program counter is correlated with t. An update operation of the execution code works with an update component that also contains a new code term t_u and a map between the possible program counters in the old term t and new program counters in t_u. The execution of the update operation would then replace the execution term with the new one, and the continuing code would be the one given by the new term t_u and the associated program counter given by the map.

References

[1] E. B. Johnsen, O. Owe, and I. Simplot-Ryl. A Dynamic Class Construct for Asynchronous Concurrent Objects. In *FMOODS*, volume 3535 of *LNCS*, pages 15–30. Springer, 2005.

[2] P. D. Mosses. Foundations of Modular SOS. In *MFCS*, volume 1672 of *LNCS*, pages 70–80. Springer, 1999.

[3] P. D. Mosses. Modular structural operational semantics. *J. Log. Algebr. Program.*, 60-61:195–228, 2004.

[4] G. D. Plotkin. A structural approach to operational semantics. Lecture Notes DAIMI FN-19, Univ. Aarhus, 1981.

[5] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *POPL*, pages 183–194. ACM, 2005.

[6] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.

[7] I. C. Yu, E. B. Johnsen, and O. Owe. Type-Safe Runtime Class Upgrades in Creol. In *FMOODS*, volume 4037 of *LNCS*, pages 202–217. Springer, 2006.

Petri Nets in Bigraphs Revisited *

Wusheng Wang[1], Gian Perrone[2] and Thomas Hildebrandt[2]

[1] School of Electronics Engineering and Computer Science, Peking University
wws@pku.edu.cn
[2] IT University of Copenhagen
{gdpe,hilde}@itu.dk

We briefly describe a very direct representation of general weighted Petri nets [10] as an instance of the Bigraphical Reactive Systems meta-model [4, 7, 6]. The representation improves and generalizes Milner's representation of 1-safe Petri net as bigraphs [7] to allow for general, weighted Petri nets using only a finite set of controls and reaction rules. The bigraphical Petri net has been proven correct and been implemented and tested using the bigraphical editor Big Red[1] and model checker BigMC [9]. The work is part of ongoing work on developing and implementing techniques for constructing, combining and relating Bigraphical Reactive Systems representing formal process languages such as Petri nets and π-calculus, as well as standard industrial modelling languages such as e.g., UML [5, 2] and BPMN [1]. We assume that the readers are familiar with Petri nets and will omit and not rely on detailed knowledge of Bigraphical Reactive Systems.

Petri nets are likely the earliest, the most well-known, and the most-used formalism for the description and analysis of distributed concurrent processes. They have had a major influence on current industrial modelling standards such as UML Activity Diagrams and Business Process Modelling Notation (BPMN). A key reason for the popularity is the very simple, yet expressive and fully formalised, core model with an intuitive graphical notation describing not only distributed processes, but also their run-time state.

The population explosion within both the Petri net and the process calculi families clearly illustrates that it is very unlikely to find a single process model that fits all purposes; modelling happens at different levels of abstraction and the nature of what we want to model changes over time. What we need — and could hope to find instead — is a meta-model and theoretical framework that facilitates some degree of unification of the many different models and reasoning techniques, as well as the continued evolution and experimentation with new process languages. Ideally, the framework should also encompass the industrial standards and allow techniques for refining abstract specifications into concrete implementations.

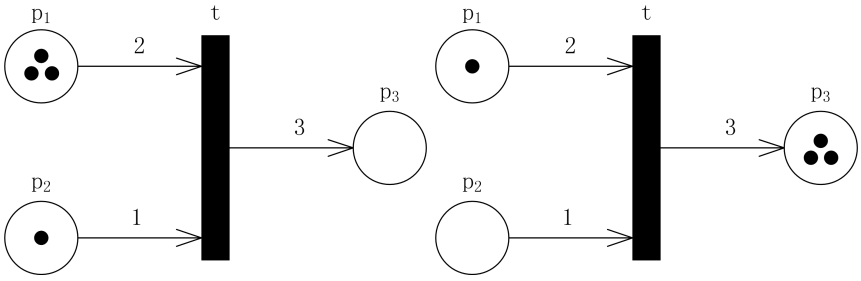
The Bigraphical Reactive Systems meta-model was introduced by Milner with the goal of providing such a meta-framework. Briefly, the Bigraphical Reactive Systems meta-model is a graph-rewriting framework in which the graphs, named bigraphs, generalize term graphs of process calculi with shared names. Concretely, a bigraph consists of two graphs, the *place graph* and the *link graph*. The place graph is a (tuple) of trees which can convieniently represent abstract syntax trees of a process calculus or programming language, the nodes of a graphical modelling language and nested virtual or physical locations. The link graph represents sharing of names, i.e. capturing *connectivity* or references, between the nodes of the place graph.

A Bigraphical Reactive System instance, just referred to as a BRS, is defined by providing a signature defining the set of types of nodes in the place graph and a set of reaction rules, which model the dynamic behaviour.

*This research work is supported by the Jingling Genie project, funded by the Danish Council for Strategic Research in Denmark, 2009 - 2013.

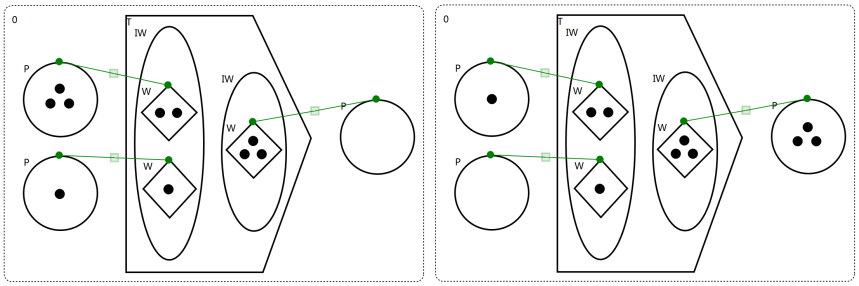
[1] Big Red [3] is available from http://bigraph.org/papers/gcm2012/.

In addition to representations of the π-calculus, Milner has demonstrated that bigraphs also allow a very direct presentation of both the static and dynamic structure of 1-safe Petri nets, using a unique control, i.e. type of node, and reaction rule for each possible number of incoming and outgoing arcs from a transition. While his approach trivially generallises to general, weighted Petri nets it suffers in practice from needing an infinite signature and an infinite set of rules in order to allow any arity of incoming and outgoing arcs and weights on transitions and markings.



(a) Petri net with marking

(b) Petri net after transition is fired



(c) The bigraphical representation of Fig. 1(a) in Big Red

(d) The bigraphical representation of Fig. 1(b) in Big Red

Figure 1: Weighted Petri net and its bigraphical representation, before and after firing the transition.

Fig. 1 shows a simple, weighted Petri net with a single transition and three places before and after firing the transition, together with the bigraphical representation in our BRS, created in the Big Red editor.

The signature of the BRS includes the controls $\{P, T, IW, OW, W, Token\}$ representing respectively the places (P) , transitions (T), sets of input weights (IW) and output weights (OW), weights (W), and tokens (Token). The Big Red editor by default labels nodes by their controls in the graphical layout, but supports suppressing the label and definition of different shapes for each control to allow visual distiction of the different controls. In Fig. 1(c) and Fig. 1(d) places are circles labelled with the control P, transitions are boxes labelled with the control T and with an extra corner indicating the output end. Each transition contains a set of input weights and a set of output weights, represented by ovals and labelled with the controls (IW) and (OW) respectively. A weight is represented by a diamond labelled with the control (W) containing as many nodes with control Token (represented by a black dot and no label) as the size of the weight. Similarly, the marking of a place is (as in the standard graphical representation of Petri nets shown in Fig. 1(a) and Fig. 1(b)) represented by a number of nodes with control Token.

The link graph is indicated by the green edges, allowing to relate the weights to places.

Fig. 1(c) and Fig. 1(d) only show how the static structure is represented. The dynamics is represented by bigraphical reaction (rewrite) rules. There is not space for showing all the reaction rules here. The key idea is to have a rule that copies a transition (and its weights) to a temporary transition, with control TCheckEn, which is used during the check for enabledness. The check for enabledness is done by removing tokens from places and weights pairwise until either all weight in TCheckEn are empty or one of the places gets empty before the weight is empty. In the first case, the transition can fire, which is handled by a rule copying tokens from output weights to the corresponding output places and finally removing the temporary transition. In the latter case, the transition is not enabled and the tokens removed from places are copied back to the input places and the temporary transition is deleted.

Conclusion and Future Work We have outlined a BRS representations of general, weighted Petri nets, improving on Milner's representation of 1-safe nets by allowing any number of tokens in places and only needing a finite signature. The BRS representation has been proven correct and has been tested and implemented using the Big Red editor and the BigMC model checker, which has helped us validate the representation experimentally. As future work we aim to explore if our approach could be transferred to other graph and term rewriting frameworks, such as Maude. We believe this is a good starting point for exploring more complex variants of Petri net as BRS, including time, as well as the BPMN modelling language. Indeed, the work is part of an ongoing effort developing techniques to allow for constructing, combining, refining [8] and relating BRS variants of formal process languages such as Petri net and π-calculus as well as standard industrial modelling languages such as e.g. UML Activity Diagrams and BPMN.

References

[1] Business process model and notation (BPMN). *At the BPMN website http://www.bpmn.org*, 2011.

[2] Marlon Dumas and Arthur ter Hofstede. UML activity diagrams as a workflow specification language. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 76–90. Springer Berlin / Heidelberg, 2001.

[3] A. Faithfull, G. Perrone, and T. Hildebrandt. Big red: A development environment for bigraphs. In *Fourth International Workshop on Graph Computation Models*, Bremen, 2012. To appear.

[4] O.H. Jensen and R. Milner. Bigraphs and mobile processes (revised). *Technical report, University of Cambridge Computer Laboratory*, 2004.

[5] J.P. López-Grao, J. Merseguer, and J. Campos. From UML activity diagrams to stochastic Petri nets: application to software performance engineering. *ACM SIGSOFT software engineering notes*, 29(1):25–36, 2004.

[6] R. Milner. Pure bigraphs. *University of Cambridge, Tech. Rep. UCAM-CL-TR-614*, 2005.

[7] Robin Milner. Bigraphs for Petri nets. In Jrg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 161–191. Springer Berlin / Heidelberg, 2004.

[8] G. Perrone, S. Debois, and T. Hildebrandt. Bigraphical refinement. In John Derrick, Eerke A. Boiten, and Steve Reeves, editors, *Refine*, volume 55 of *EPTCS*, pages 20–36, 2011.

[9] Gian Perrone, Søren Debois, and Thomas T. Hildebrandt. A model checker for Bigraphs. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1320–1325, 2012.

[10] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universitt Hamburg, 1962.

Soundness of a Reasoning System for Asynchronous Communication with Futures

Crystal Chang Din and Olaf Owe

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

Distributed systems play an essential role in society today. For example, distributed systems form the basis for critical infrastructure in different domains such as finance, medicine, aeronautics, telephony, and Internet services. It is of great importance that such systems work properly. However, quality assurance of distributed systems is non-trivial since they depend on unpredictable factors, such as different processing speeds of independent components. It is highly challenging to test such distributed systems after deployment under different relevant conditions. These challenges motivate frameworks combining precise modeling and analysis with suitable tool support. In particular, *compositional verification systems* allow the different components to be analyzed independently from their surrounding components. Thereby, it is possible to deal with systems consisting of many components.

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [12]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [2]. Concurrent objects communicating by *asynchronous method calls*, which allows the caller to continue with its own activity without blocking while waiting for the reply, combine object-orientation and distribution in a natural manner, and therefore appears as a promising paradigm for distributed systems. Moreover, the notion of *futures* [4, 15, 10, 16] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results.

ABS is a high-level imperative object-oriented modeling language, based on the concurrency and synchronization model of *Creol* [14]. It supports futures and concurrent objects with an asynchronous communication model suitable for loosely coupled objects in a distributed setting. In *ABS*, each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing. The concurrent object model of *ABS* without futures supports compositionality because there is *no direct access* to the internal state variables of other objects, and a method call leads to a new process on the called object. With futures, compositionality is more challenging.

In this paper, we focus on the communication model of *ABS* with futures but ignore the aspects of inheritance and object creation. A compositional reasoning system for *ABS* with futures has been presented in [9] based on local communication histories. We here show that this system is sound with respect to a revised version of the operational semantics of *ABS*, which incorporates a notion of global communication history.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [5, 11]. At any point in time the communication history abstractly captures the system state [7, 6]. In fact, traces are used in the semantics for full abstraction results (e.g., [13, 1]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of

possible histories, expressing safety properties [3].

In our reasoning system, we formalize object communication by an operational semantics based on four kinds of communication events [8], capturing shared futures, where each event is visible to only one object as shown in Fig. 1. Consequently, the local histories of two different objects share no common events. For each object, a history invariant can be derived from the class invariant by hiding the local state of the object. Modularity is achieved since history invariants can be established independently for each object, without interference, and composed at need. This results in behavioral specifications of dynamic system in an open environment. Such specifications allow objects to be specified independently of their internal implementation details, such as the internal state variables. In order to derive a global specification of a system composed of several components, one may compose the specification of different components. Global specifications can then be provided by describing the observable communication history between each component and its environment.

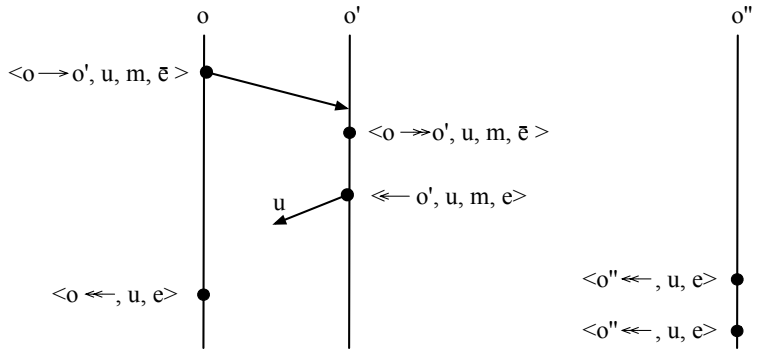


Figure 1: A method call cycle: object o calls a method m on object o' with *future* u. The four kinds of events are indicated by $\rightarrow, \twoheadrightarrow, \leftarrow$, and $\twoheadleftarrow$, indicating method invocation, method initiation, writing to future and reading from future, respectively. The events on the left-hand side are visible to o, those in the middle are visible to o', and the ones on the right-hand side are visible to o''. There is an arbitrary delay between message receiving and reaction.

The main contribution of this paper is the proof of soundness with respect to the revised operational semantics including a global communication history. The operational semantics is implemented in Maude by rewriting rules and can be exploited as an executable interpreter for the language, such that execution traces can be automatically generated while simulating programs. In earlier work [8, 9], a similar proof system is derived from a standard sequential language by means of a syntactic encoding. However, soundness with respect to the operations semantics was not considered. A challenge of the current work is that the presence of shared futures complicate compositional reasoning and also the soundness proof. We therefore focus the current work on the *ABS* communication model with futures, and ignore other aspects such as object creation.

An *ABS* reasoning system is currently being implemented within the KeY framework at Technical University Darmstadt. With the tool support from KeY for (semi-)automatic verification, it will be valuable to verify larger *ABS* programs. An elevator system has been implemented, simulated and tested according to the generated histories. The elevator system will be used here to illustrate the language and the reasoning system.

References

[1] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(7):491–518, 2009.

[2] A. Ahern and N. Yoshida. Formalising Java RMI with explicit code mobility. *Theoretical Computer Science*, 389(3):341 – 410, 2007.

[3] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, Oct. 1985.

[4] H. G. Baker Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[5] M. Broy and K. Stølen. *Specification and Development of Interactive Systems.* Monographs in Computer Science. Springer, 2001.

[6] O.-J. Dahl. Object-oriented specifications. In *Research directions in object-oriented programming*, pages 561–576. MIT Press, Cambridge, MA, USA, 1987.

[7] O.-J. Dahl. *Verifiable Programming.* International Series in Computer Science. Prentice Hall, New York, N.Y., 1992.

[8] C. C. Din, J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3):227–256, 2012.

[9] C. C. Din, J. Dovland, and O. Owe. Compositional reasoning about shared futures. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. International Conference on Software Engineering and Formal Methods (SEFM'12)*, volume 7504 of *LNCS*, pages 94–108. Springer, 2012.

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

[11] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice Hall, 1985.

[12] International Telecommunication Union. Open Distributed Processing - Reference Model parts 1–4. Technical report, ISO/IEC, Geneva, July 1995.

[13] A. S. A. Jeffrey and J. Rathke. Java Jr.: Fully abstract trace semantics for a core Java language. In *Proc. European Symposium on Programming*, volume 3444 of *LNCS*, pages 423–438. Springer, 2005.

[14] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[15] B. H. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In D. S. Wise, editor, *Proc. SIGPLAN Conference on Programming Language Design and Implementation (PLDI'88)*, pages 260–267. ACM Press, June 1988.

[16] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86). Sigplan Notices*, 21(11):258–268, Nov. 1986.

Compositional Analysis of Resource Bounds for Software Transactions

Thi Mai Thuong Tran*, Martin Steffen, and Hoang Truong

Dept. of Computer Science, University of Oslo, Norway and University of Engineering and Technology, Vietnam National University of Hanoi

1 Motivation

Software Transactional Memory (STM) has recently been introduced to concurrent programming languages as an alternative for locked-based synchronization. STM enables an optimistic form of synchronization for shared memory. Each transaction is free to read and write to shared variables and a log is used to record these operations for validation or potentially rollbacks at commit time. Maintaining the logs is a critical factor of memory resource consumption of STM.

One of the advanced transactional calculi recently introduced is Transactional Featherweight Java (TFJ) [2], a transactional object calculus which supports *nested* and *multi-threaded* transactions. Multi-threaded transactions mean that inside one transaction there can be more than one thread running in parallel. *Nested* means that inside one transaction, there can be another transaction nested. Furthermore, nested transactions must commit before their parent transaction, and if a parent transaction commits, all threads spawned inside a transaction must join via a commit.

The convenience of transactional programming comes at a cost; in particular each transaction needs a local copy of the part of the memory, in this case the heap, it accesses. Furthermore, in order to be able to roll-back, a transaction may need to keep different versions of a memory location (sometimes called a log). The entaild resource consumption may lead to a memory overrun in the following way:

- duplicating parent transactions for the conflict checking. Each time a new thread is spawned, the log of its parent transaction is *copied* into the spawned thread's log. I.e., a spawned thread will "inherit" its parent transactions. So the resources for the new thread need to be calculated to store information in the parent transaction's log apart from its own log.
- a certain amount of transactions run in parallel at the same time which will increase the overall number of transactions in the system.
- The number of different versions of a memory location, i.e., the length of the log, contributes to memory consumption.

In this work, we will statically predict resource consumption in connection with transactions, taking into account the maximum number of logs produced at any

* E-mail: tmtran@ifi.uio.no

$P ::=$	$\mathbf{0} \mid P \parallel P \mid p\langle e \rangle$	processes/threads
$L ::=$	$\mathsf{class}\ C\{\boldsymbol{f}{:}\boldsymbol{T}; K; \boldsymbol{M}\}$	class definitions
$K ::=$	$C(\boldsymbol{f} : \boldsymbol{T})\{\mathsf{this}.\boldsymbol{f} := \boldsymbol{f}\}$	contructors
$M ::=$	$m(\boldsymbol{x}{:}\boldsymbol{T})\{e\} : T$	methods
$e ::=$	$v \mid v.f \mid v.f := v \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{let}\ \ x{:}T = e\ \texttt{in}\ \ e \mid v.m(\boldsymbol{v})$	expressions
	$\mid\ \texttt{new}\ \ C(\boldsymbol{v}) \mid \texttt{spawn}\ \ e \mid \texttt{onacid} \mid \texttt{commit}$	
$v ::=$	$r \mid x \mid \texttt{null}$	values

Table 1. Abstract syntax

given point in parallel execution of transactions, and the length of transactional executions, in this way refining our earlier results on resource consumption. Furthermore, our analysis is *compositional* in particular wrt. parallel composition, which is an improvement over [3].

2 A type and effect system for a transactional calculus

Syntax

The language used in this paper is, with some adaptations, taken from [2] and a variant of Featherweight Java (FJ) [1] extended with *transactions* and a construct for thread creation. The syntax of our calculus is given in Table 1. The main adaptations are: we added standard constructs such as sequential composition (in the form of the let-construct) and conditionals.

The language is multi-threaded: `spawn` e starts a new thread of activity which evaluates e in parallel with the spawning thread. Specific for TFJ are the two constructs `onacid` and `commit`, two dual operations dealing with transactions. The expression `onacid` starts a new transaction and executing `commit` successfully terminates a transaction.

Typing judgment

In order to estimate the maximal resource consumption used by an expression in the program, we introduce the judgments of the expressions as follows:

$$n_1 \vdash e :: n_2, h, l, t, S \tag{1}$$

The elements n_1, n_2, h, and l are natural numbers with the following interpretation. n_1 and n_2 are the pre- and post-condition for the expression e, capturing the nesting depth: starting at a nesting depth of n_1, the depths is n_2 after termination of e. We call the numbers n_1 resp. n_2 also the current balance of the thread. Starting from the pre-condition n_1, the numbers h and l represent the maximum resp., the minimum value of the balance during the execution of e (the "highest" and the "lowest" balance during execution). The numbers so

far describe the balances of the thread executing e. During the execution of e, however, new child threads may be created via the spawn-expression and the remaining elements t and S take (also) their contribution into account. The number t represents the maximal, overall ("total") resource consumption during the execution of e, including the contribution of all spawned threads. The last component S is a multiset of pairs of natural numbers, i.e., it is of the form $\{(p_1, c_1), (p_2, c_2), \ldots\}$. For all spawned threads, S keeps its maximal contribution to the resource consumption at the point after e, i.e., (p_i, c_i) represents that the thread i can have maximally a resource need of $p_i + c_i$, where p_i represents the contribution of the spawning thread ("parent"), i.e., the current nesting depth at the point when the thread is being spawned, and c_i the additional contribution of the child threads itself.

3 Main results

- We present a concurrent object-oriented calculus supporting nested and multi-threaded transactions. The language features non-lexical starting and ending of multi-threaded and nested transactions.
- We propose a type and effect system to estimate the upper bound of resource consumption during the program's execution.
- We provide an *inference* system, avoiding user-provided annotations to specify the resource consumption up-front.
- We show the soundness of the static analysis.

References

1. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, pages 132–146. ACM, 1999. In *SIGPLAN Notices*.
2. S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, Aug. 2005.
3. T. V. Xuan, H. T. Anh, T. Mai Thuong Tran, and M. Steffen. A type system for finding upper resource bounds of multi-threaded programs with nested transactions. In *ACM Proceedings of the 3rd ACM International Symposium on Information and Communication Technology SSoICT*. ACM, Aug. 2012.

Maximizing Entropy over Markov Processes*

Fabrizio Biondi[1], Axel Legay[2], Bo Friis Nielsen[3] and Andrzej Wąsowski[1]

[1] IT University of Copenhagen, Denmark
[2] INRIA Rennes, France
[3] Technical University of Denmark, Lyngby, Denmark

Context. Consider a common authentication protocol: a user is requested to enter a password. She is granted access to the system if the password corresponds to the one stored in the system. The goal of the protocol is to refuse access to users that do not know the password (the secret). The protocol should also prevent an attacker from guessing or learning the secret, by interacting with the system. We want to decide whether the protocol is secure or not against an attacker that wants to access it, but does not know the password. The first question though is: what does it mean for a system to be secure?

One possible answer is that a system is secure if the attacker cannot authenticate in the system. This however cannot be guaranteed. There is a non-zero probability that an attacker is lucky and simply guesses the password. We could require that the attacker cannot learn the password by interacting with the system. However, a password is not an atomic object: would we consider the system secure if the attacker could learn 90% of the password? Finally, most-strictly, we could require that in a safe authentication protocol no attacker can learn anything about the password through interaction, the so called non-interference definition. Unfortunately, this definition usually breaks in practice: if the attacker inserts a random password and gets rejected he will learn that the password is not the one he tried, breaking non-interference [8].

In practice it is not possible to give a qualitative, yes-no definition of security that is not overly strict (rejects any authentication protocol) or overly permissive (accepts protocols leaking a significant amount of the password). The core problem is that even a secure protocol will leak a small amount of information about the password [4]. Quantitative analysis techniques have been developed to precisely quantify this amount, leaving to the analyst to decide whether it is acceptable for the protocol analyzed.

Quantitative Information Flow [1] is a quantitative approach to compute the number of bits of information an attacker would gain about the confidential data of a system by interacting with the system and observing its behavior.

The difference between the information [9] of a given attacker about the secret before and after a single attack is called *leakage*. Different attackers would leak different amount of information. A possible quantitative definition of security of a system is the maximum leakage of information (so leakage towards the attacker that can leak most). Maximum leakage is known in security theory as *channel capacity*. As said, no single attack can leak an amount of information higher than the system's channel capacity [6]. For a deterministic system, the leakage is the entropy of the observable behavior of the system conditioned by the attacker's behavior [5], and thus computing the channel capacity reduces to computing the behavior of the system that maximizes entropy. This security guarantee is only valid for one process; any perturbation of the process' behavior would invalidate it.

Problem. This technique evaluates only the security of a completely defined system, since it needs to evaluate the likelihood and estimate the loss of confidential data of each possible user behavior and system's reaction to it. Alas, when a security protocol is written the designer

*The research presented in this presentation has been partially supported by MT-LAB, a VKR Centre of Excellence for the Modelling of Information Technology.

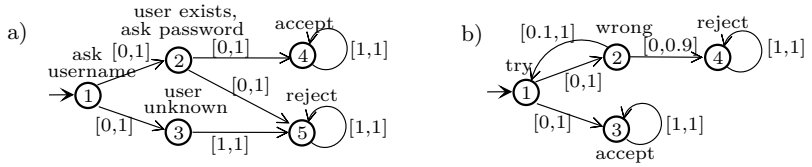


Figure 1: a) Two-step Authentication b) Repeated Authentication

needs the freedom to leave some parts underspecified, like the size of a password or the number of usernames in the system. An underspecified system can be seen as the infinite union of all its implementations. Providing a security guarantee for it is challenging, as it requires computing the maximum leakage over all attacks by all possible attackers to all possible implementations.

Contribution. We will present a method to obtain a safety guarantee for a protocol specification when it is possible and signal that the protocol is unsafe otherwise. To the best of our knowledge, no other approach to channel capacity computation for infinite classes of processes exists.

To execute a formal analysis of a process specification we need models for the processes and specifications and procedures to analyze them. We use Markov chains (MCs) as process models, with the states of the a MC representing the observable components of the system and the transition probabilities raising from the attacker's uncertainty about the secret. A single MC represents an attack scenario, in which a given attacker interacts with and observes a given deterministic system. The entropy of the MC for the scenario is the leakage from the system to the attacker. We present a polynomial-time procedure to compute entropy for a Markov chain, by reduction to computation of the Expected Total Reward [7, Chpt. 5] of a local non-negative reward function associated with states over the infinite horizon.

We present theories and algorithms to synthesize the process with maximum entropy among all those respecting a given probabilistic specification, allowing us to give a security guarantee valid for all the infinite processes respecting the specification. To encode the infinite number of possible attackers and implementations we want to consider we need the continuity of real-valued transition probability intervals, so we use and Interval Markov Chains (Interval MCs) [3] as specification models.

Interval MCs allow us to generalize the scenario to all possible attackers interacting with all possible implementations of a specification. The Maximum Entropy resolution of the nondeterminism in the Interval MC is thus the scenario with the greatest leakage, and its entropy is the channel capacity of the specification and the maximum amount of information that can be leaked by any single attack to any given implementation of the specification.

We show that in general such Maximum Entropy implementation of an Interval MC may not exist, as implementations may have infinite or unbounded entropy. We characterize these cases and provide a polynomial-time procedure for deciding finiteness and boundedness of the entropy of all implementations of an Interval MC by using an extended notion of end components [2].

Even when the Maximum Entropy implementation exists, computing it requires to solve a multidimensional nonlinear maximization problem on convex sets; we present a numerical procedure to synthesize an implementation maximizing a given reward function on Interval MCs with arbitrary approximation [10] and use it to obtain an arbitrary approximation of the Maximum Entropy implementation.

Illustrative Examples. Consider two examples of models of deterministic authentication processes. Figure 1a presents an Interval MC for a two-step authentication protocol with username and password. The actual transition probabilities will depend on how many usernames exist in

the system, on the respective passwords, on their length and on the attacker's knowledge about all of these; but staying at the specification level allows us to consider the worst case of all these possible combinations, and thus gives an upper bound on the leakage. The Maximum Entropy implementation for the Two-step Authentication is given in Fig. 2; its entropy is the channel capacity of the system over all possible prior informations and behaviors of the attacker and design choices of the implementer.

Consider another example. Figure 1b presents an Interval MC specification of the Repeated Authentication protocol, in which the user inserts a password to authenticate, and is allowed access if the password is correct. If not, the system verifies if the password entered is in a known black list of common passwords, in which case it rejects the user, considering it a malicious attacker. If the password is wrong but not black listed the user is allowed to try again. The black list cannot cover more than 90% of the possible passwords.

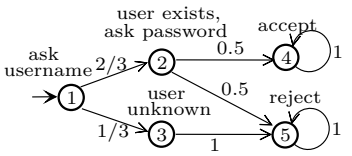


Figure 2: Maximum Entropy implementation for the Two-step Authentication

Note that the transition probabilities from state 2 is a design choice left to the implementer of the system, while the transition probabilities from state 1 depend on a given attacker's knowledge about the password, and thus cannot be controlled even by the implementer. By abstracting these two different sources of nondeterminism at the same time we maximize entropy over all possible combinations of design choices and attackers, effectively finding the channel capacity of the specification.

The implementation maximizing entropy is the one revealing most information about the system's secret. This is consistent with our intuition. For instance, if the black list is empty the user can continue guessing the password indefinitely: the probability of eventually reaching state 3 is 1. In this implementation sooner or later the attacker will discover the password, and thus the system's secret will be completely revealed.

In fact, it is not possible to give a Maximum Entropy implementation for such protocol; this means that whatever is the length n of the secret, it is possible to give an implementation that leaks all n bits of information. We show how these undesirable cases can be recognized in polynomial time and how the specification can be modified to avoid them.

References

[1] David Clark, Sebastian Hunt, and Pasquale Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 15, 2007.

[2] Luca de Alfaro. *Formal Verification of Probabilistic Systems.* PhD thesis, Stanford, 1997.

[3] Bengt Jonsson and Kim Guldstrand Larsen. Specification and refinement of probabilistic processes. In *LICS*, pages 266–277. IEEE Computer Society, 1991.

[4] Pasquale Malacaria. Assessing security threats of looping constructs. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '07, pages 225–235, New York, NY, USA, 2007. ACM.

[5] Pasquale Malacaria. Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. *CoRR*, abs/1101.3453, 2011.

[6] Jonathan K. Millen. Covert channel capacity. In *IEEE Symposium on Security and Privacy*, pages 60–66, 1987.

[7] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley-Interscience, April 1994.

[8] P. Ryan, J. McLean, J. Millen, and V. Gligor. Non-interference, who needs it? In *Computer Security Foundations Workshop, 2001. Proceedings. 14th IEEE*, pages 237 –238, 2001.

[9] Claude E. Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27:379–423, July 1948.

[10] J. Stoer, R. Bulirsch, R. Bartels, W. Gautschi, and C. Witzgall. *Introduction to Numerical Analysis*. Texts in Applied Mathematics. Springer, 2010.

Literals in programming languages

Magne Haveraaen

BLDL, Institutt for informatikk
University of Bergen

1 Overview

Most programming languages provide a large selection of literals for the ease of the programmer. Literals typically include integers, floating numbers, characters and character strings. In typed programming languages such as C, C++ and Java, there may be a multitude of integral and floating types. In order for the compler to figure out how to handle simple constant expressions involving literals, there often is a literal for each of these, 123l for a long constant, 123 for an int constant, 123.0 for a double constant, 123.0f for a single presision float, etc. Then a programmer may write 2147483647l+2147483647l and expect 4294967294l, as opposed to -2 for the 32bit int sum 2147483647+2147483647.

This gives the programmer control, but leads to a plethora of numerical literals and conversion rules which may utterly confuse the novice.

Here we attempt to simplify this by introducing a separate algebra for each of the literal types integer, floating, character and string. By making these initial algebras, we get a unique homomorphism from literal expressions to the computer types, allowing the compiler to delay conversion from the literal type until assignment to a computer type.

2 Initial unit ring

The standard logarithmic notation for literal integers, the decimal position system, is based on the fact that integers are the initial unit ring. This allows us to use the ring operations (addition and multiplication) to multiply the digit value by its position, then add these intermediate results to achieve the integer value denoted by the literal. Negative literals apply the ring negation operation to the result.

This means there is a unique homomorphism from any literal integer expression using the ring operations, to any ring data type. The evaluation of the expression can take place at the literal integer level, or at the target algebra level. In the former case, the compiler may use the standard digit based algorithms for computation on the literals. In the latter case, the compiler may use the computer instructions to compute the value. In either case the result will be the same.

For example, the signed 8bit integer, "byte" in Java speak, value -127 can be computed from 100+29 either from (byte)(129) or (byte)100+(byte)29.

This also makes clear that when we use functions not among the ring operations, such as the absolute value function "abs", then we cannot benefit from this approach. Consider the expressions (byte)abs(-129)=(byte)129=-127 versus abs((byte)-129)=abs(127)=127.

Care also needs to be taken when considering iteration of the ring operations. For instance, $(byte)\sum_{i=1}^{n} p = (byte)(n * p) = (byte)n * (byte)p$ according to the ring homomorphism, but $(byte)\prod_{i=1}^{n} p = (byte)(p^n) = ((byte)p)^n$ which in general is different from $((byte)p)^{(byte)n}$.

We can of course explore this for other ring data types than the integral data types. Let M be the ring of all 3*3 matrices with matrix addition and matrix multiplication as the ring

operations. Then 1 would yield the unit diagonal matrix, and 2*3 would be the diagonal matrix with 6 on the diagonal. Now for N, the ring of all 3*3 matrices with matrix addition and Hadamard-product as the ring operations, 1 would yield the matrix containing only ones, and 2*3 would be the matrix containing only the number 6.

3 Characters and Strings

A character used to be formalised by machine specific character sets, e.g., ASCII, Norwegian ASCII and EBCDIC. Now significant international collaboration on the Unicode encoding, means that in principle any character or symbol normally used in any language can be presented as a character on a computer.

Strings are then the free monoid over the characters.

Normally a programming language has only one character type and one string type. The initial algera technique does not give any extra advantage in this case-

4 Floating Point Numbers

Floating point numbers, such as float, double, long double, are approximations to field algebras. The normal notation allows literals as integrals with a decimal scaling factor (by multiplication or division). The use of these literals in expressions have problems with operators beyond the field operations. This is much the same problem as we saw for the unit ring and integral types.

A finer module system for the ABS language

Elmo Todurov and Keiko Nakata

Institute of Cybernetics at TUT, Tallinn, Estonia, elmo@ioc.ee, keiko@cs.ioc.ee

1 Introduction

Modular programming is the practice of splitting program code into smaller components, or *modules*. Each module deals with a single task, with its module interface describing the functionality of the module. Historically [2], modularity arose from separate compilation, a way to optimize compilation speeds—it takes some seconds to compile a single .c file on modern machines, and it takes approximately an hour to compile the whole Linux kernel. Thus separate compilation makes software development much easier by shortening the development cycle which involves repeated (re)compilation. Today, however, modularity is mainly used as a program design strategy. Modules are seen as a way to impose order on programs, in the sense that conceptually related code should be grouped together. Modules may import some functionality—functions, datatypes and possibly also data values—from other modules and export their own functionality for other modules. If the intended behavior of the modules is well enough specified, it is easy to split programming work between people or teams.

Different module systems have different features. Module systems should ideally allow as many modularity supporting features as possible. Namespace management is probably the most common feature: names defined in one module should not be visible in other modules, unless explicitly exported. Modules should be pluggable: a module can be replaceable by another module, assuming that their interfaces match, without recompilation. This would allow for easy exchange of debug and optimized builds, for instance, or versions optimized for different criteria [1]. For example, Java uses globally unique module names, which prevent any drop-in replacements. Abstract (or opaque) types also enhance modular design. They can be used to hide implementation details: values of abstract types can only be manipulated through interface functions. A module system should provide for easy separate compilation. Moreover, modules could take parameters like functors in OCaml and could be hierarchical [1].

In this work, we design a module system for a Java-like class-based object-oriented language, but without implementation inheritance. More precisely, the language we consider is based on ABS, a language developed in the HATS project [3]. The goal of this work is to present a finer module system for ABS. Currently ABS supports a Haskell-like simple module system. In this paper, we refine it to support separate typechecking and pluggable modules via explicitly declared interfaces, and abstract types.

2 Example

The example in Figure 1 presents an associative array. An associative array is a data structure that associates keys of a given type with values of a given other type.

For us, a module consists of import and export declarations and implementation. In Figure 1, the module $AssocArray$ imports a type of the keys $IKey$ and a type of the values $IValue$. It also requires the key type to have a comparator function. It exports the class $AssocArray$ along with the interface $IAssocArray$, specifying that the $AssocArray$ class implements associated arrays for the imported key and value types. An interface can be *abstract* in import

Figure 1: AssocArray module

```
import: abstract interface IKey { Bool lessThan(IKey) }
        abstract interface IValue {}
export: abstract interface IAssocArray { Unit set(IKey, IValue)
                                          IValue get(IKey) }
        class AssocArray implements IAssocArray
implementation: interface IAssocArray { Unit set(IKey, IValue)
                                        IValue get(IKey)
                                        Unit balanceTree() /* hidden */ }
                class AssocArray { Unit set(IKey, IValue) {...}
                  IValue get(IKey) {...}; Unit balanceTree() {...} }
```

or export. Abstract interfaces in export can have less methods than the corresponding version in module implementation, thus hiding information. Correspondingly, abstract interfaces in import signify that not all methods of the interfaces are visible for this module. Abstract interfaces enable modules to export values abstractly: values implementing abstract interfaces can only be manipulated through interface functions. This can be used to enforce module-wise invariants, as abstract values cannot be tampered with outside the module. For instance, in Figure 1, in the implementation of *AssocArray* module, the interface $IAssocArray$ has one more method $balanceTree()$ which keeps the tree balanced. This balancing method should not be called from outside. This is good for information hiding: the users of the associative array should not know how it is implemented.

To use the *AssocArray* module, one has to link it against a module that exports the $IKey$ and $IValue$ types, then the resulting linked module exports the associative array for the other modules to use.

3 Type System

Our type system is based on Cardelli's type system for modules [2] and Featherweight Java [4]. The main type judgement $S \vdash M \ : \ E$ is OK states that a module (with implementation M) is well-typed, exporting E and using S as imports from other modules. The typing rules are mostly standard, except that we have to forbid inheriting from and instantiating abstract interfaces, which would make the type system unsound. For instance, in Figure 2, the module M exports the abstract interface A whose implementation has the $aux()$ method. $aux()$ is not exported. Inside M one may call $aux()$. The module N imports the abstract interface A and defines a class U implementing A. The class U lacks the $aux()$ method. Indeed, if one links M against N, the method $ABuse()$ will produce a type error.

Soundness Typically, type soundness means that if a program is well-typed, then it will not have runtime type errors. For our type system, however, soundness guarantees that separately typechecked modules with compatible interfaces can be safely linked [2] to form a complete program.

Theorem 1. *If* $\bigcup_i S_i \subseteq \bigcup_i E_i$ *and for all* i, j, $i \neq j \Rightarrow E_i \cap E_j = \emptyset$ *and for all* i, $S_i \vdash M_i \ : \ E_i$ *is OK, then* $\emptyset \vdash \sum_i M_i \ : \ \sum_i E_i$ *is OK.*

Figure 2: Abstract interfaces: module M

```
/* Module M */
import: /* empty */
export: abstract interface A { A create() }
        abstract interface B { Unit use(A) }
        class Cb implements B
module: interface A { A create(); Unit aux() }
        interface B { Unit use(A) }
        class Cb implements B { Unit use(A x) { x.aux() } }

/* Module N */
import: abstract interface A { A create() }
        abstract interface B { Unit use(A) }
        class Cb implements B
export: /* empty */
module: class U implements A {
          A create() { ... }
          ABuse() { c = new Cb(); c.use(this.create()) /* type error */ } }
```

Let a program $\{(S_i, E_i, M_i)\}$ be a set of modules, each of which consists of import declarations S_i, export declarations E_i and implementation M_i. The theorem states that if the program is closed (all imports are matched by some exports), $\bigcup_i S_i \subseteq \bigcup_i E_i$, and if exports have no conflicts, $\forall i, j \,.\, i \neq j \Rightarrow E_i \cap E_j = \emptyset$, and if every module is well-typed separately, $\forall i \,.\, S_i \vdash M_i \,:\, E_i$ is OK, then the resulting linked module is well-typed, $\emptyset \vdash \sum_i M_i \,:\, \sum_i E_i$ is OK.

4 Conclusion

In this paper, we presented a type and module system for ABS. It supports separate compilation, pluggable modules and abstract types. The type system is proved sound. As future work, we would like to extend our module system to support parametric and hierarchical modules.

Acknowledgement This research was supported by the EU FP7 ICT project no. 231620 (HATS).

References

[1] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Trans. Program. Lang. Syst.*, 21(4):813–847, July 1999.

[2] Luca Cardelli. Program fragments, linking, and modularization. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 266–277, New York, NY, 1997. ACM.

[3] HATS. The HATS project, May 2012. http://www.hats-project.eu.

[4] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

SPECTA: A Formal Specification Language for Content Transfer Algorithms

Petter Sandvik[1,2]

[1] Department of Information Technologies, Åbo Akademi University, Finland
[2] TUCS - Turku Centre for Computer Science, Finland
petter.sandvik@abo.fi

1 Introduction

When data is transferred between nodes in a network, it is often transferred in-order. Especially in systems using the traditional client-server model, there is little need for transferring data any other way. However, in distributed systems, such as peer-to-peer networks and cloud-based systems, transferring data out of order can be advantageous, for instance by improving reliability, availability and transfer speed. For that reason, there is a need to understand different content transfer algorithms. These algorithms have previously been described in a plain text format, easily understandable by humans, or in a formalism or program code suitable for machine interpretation, which may be application specific and not always easily readable by humans. In the former case we can include BiToS [6] and DAW [3], and as for the latter case, it has previously been attempted by us [4, 5].

What we aim for is a reusable formalism that can describe the complexities of out-of-order content transfer in a simple manner, while being powerful enough to support as large a variety of distributed content transfer algorithms as possible, from peer-to-peer video streaming to cloud-based file backup. For this purpose we introduce the Specification for Content Transfer Algorithms (SPECTA) Language. In the following, we briefly overview SPECTA and present a few simple examples of using it.

2 The SPECTA Language

To be able to transfer content out of order, we need an initial ordering of the content, and a way of describing the order in which the content should be transferred. Therefore, we assume that the content to be transferred is partitioned into a finite number of pieces, enumerated sequentially by positive integers. Content transfer algorithms should then, based on properties of the pieces and of the system as a whole, step by step choose which piece to transfer next, until there are no eligible pieces left. Not all pieces are necessarily eligible at all times; for instance, while a backup solution may choose to backup the same content more than once in order to store content at different locations, a video streaming client usually does not need to transfer the same content more than once and therefore only needs to consider the pieces not already transferred. Deciding about eligibility of pieces is application-specific, and may even change over time, and in the algorithms we therefore do not specify which pieces are eligible.

When selecting the next piece to be transferred, we specify an overall condition that models a state the system must be in for the selection to take place at all, as well as criteria that describe the properties that an eligible piece must satisfy in order to be selected (1). The condition is separated from the criteria using a right triangle ($\triangleright$) symbol, and the criteria are separated from each other using a pipe (|) symbol. The use of a pipe symbol between criteria is due to its similar usage in UNIX-like command line environments, where the output of what is on the left-hand side of the pipe is used as the input for what is on the right-hand side. In SPECTA, the first criterion specifies a subset of all eligible pieces, the second criterion specifies

a subset of the first subset, and so on. In the end, we are interested in no more than one piece at a time, and therefore, if the subset consists of more than one piece after the final criterion, we assume that one piece is non-deterministically chosen from the remaining subset.

$$selection \equiv condition \triangleright criterion_1 | criterion_2 | \ldots | criterion_m, \text{ where } m \geq 1 \quad (1)$$

If the condition is not fulfilled, or the condition is fulfilled but there is no eligible piece satisfying all criteria, no piece can be selected according to (1). Therefore, we need a way of combining several selection mechanisms, for which purpose we use the semicolon (2). If the first selection is possible, it is used, but if not, the second one is tried, and so forth. Unlike the selection criteria in (1) there is no non-determinism involved if the final selection fails; we simply assume that the algorithm is designed in such a way that the correct behaviour is to do nothing until the conditions and criteria make it possible to perform a selection.

$$next = selection_1; selection_2; \ldots; selection_k, \text{ where } k \geq 1 \quad (2)$$

To express the conditions and criteria, we can use simple arithmetic operations, but we also need to define keywords and operations that can be used together with them. An important subset of these can be found in Table 1.

Main form	Other form(s)	Description
`piece`	`p`	The ID of the specific piece being considered.
`total`	`all,last`	The total number of pieces. Also the ID of the last piece, because pieces are numbered from 1.
`eligible pieces`	`elig`	The set of all eligible pieces, a subset of all pieces. The subset of eligible pieces satisfying all criteria so far.
`requested`	`r,req`	The number of pieces that have been requested.
`transferred`	`t,tr,transfered`	The number of pieces that have been transferred. This number may be larger than `requested` when transferring content in parallel.
`availability(x)`	`av(x),avail(x)`	The number of nodes that hold the piece of content specified by the parameter. If no parameter, `piece` is assumed.
`minimum(x)`	`min(x)`	The piece for which the parameter is the smallest.
`maximum(x)`	`max(x)`	The piece for which the parameter is the largest.
`random(x)`	`random(x,y)`	If the parameter is a set of pieces, returns a random piece from that set. Otherwise gives a random number from `1` to `x`, or with two parameters, in the range from `x` to `y`.
`probability(r)`	`prob(r)`	Returns `true` with the probability `r`, `0`≤`r`≤`1`.
`size(x)`		Returns the size of the piece specified by the parameter. If no parameter is given, `piece` is used.
`current`	`c,cur`	The current piece in any external use, e.g., playback position in media streaming.

Table 1: Keywords and operations

With these in mind, we can give some examples. For instance, it is possible to write an in-order transfer algorithm by specifying that the piece with the lowest number should always be selected (3). Obviously, this requires that each previously requested piece is no longer eligible.

$$next = true \triangleright min(piece) \quad (3)$$

In the piece selection method used in the original BitTorrent peer-to-peer file sharing application [2], pieces are requested randomly until one complete piece has been transferred, and after

that pieces are selected rarest-first, i.e., starting from the ones with lowest availability. The behaviour when there is more than one piece with the lowest availability is not specified [2], and therefore we leave the choice non-deterministic if the set of pieces with the lowest availability should contain more than one piece (4).

$$next = transferred < 1 \triangleright random(pieces); transferred \geq 1 \triangleright min(avail(piece)) \quad (4)$$

BiToS is a modification of BitTorrent to support streaming media [6]. This is done by partitioning the pieces into a high priority set, which consists of pieces close to being played back, and a set of pieces with lower priority. With a probability of 0.8 the rarest piece from the high priority set is chosen, otherwise the rarest piece from the low priority set is chosen. In both cases, if there is more than one piece with the same availability, the piece with the lowest piece number is chosen. The optimal size of the high probability set was found to be 8% of the complete set of pieces [6]. With SPECTA we can describe the BiToS algorithm as (5).

$$\begin{aligned} next = prob(0.8) \triangleright piece \leq current + (0.08 * total) | min(avail(piece)) | min(piece); \\ true \triangleright piece > current + (0.08 * total) | min(avail(piece)) | min(piece) \end{aligned} \quad (5)$$

As another example, consider a distributed backup solution, in which each piece of content should be distributed to three different nodes. Moreover, all content should be available on n nodes before we start distributing it to $n + 1$ nodes, and larger pieces of content should be transferred before smaller. This can be described as (6).

$$next = true \triangleright avail(piece) < 3 | min(avail(piece)) | max(size(piece)) \quad (6)$$

3 Conclusions

In this paper we have introduced SPECTA, a language for specifying content transfer algorithms. This language can be used as a starting point for generating executable code or formal models, e.g. in Event-B [1]. Moreover, we could extend our language to support more information about other nodes, such as latency, transfer speed, and uptime. This would enable not only more advanced methods of selecting what piece of content to transfer, but also specifying which nodes should be involved.

References

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

[2] B. Cohen. Incentives Build Robustness in BitTorrent. In *1st Workshop on Economics of Peer-to-Peer Systems*, June 2003.

[3] P. Sandvik and M. Neovius. The Distance-Availability Weighted Piece Selection Method for BitTorrent: A BitTorrent Piece Selection Method for On-Demand Streaming. In Antonio Liotta, Nick Antonopoulos, George Exarchakos, and Takahiro Hara, editors, *Proceedings of The First International Conference on Advances in P2P Systems (AP2PS 2009)*, pages 198–202. IEEE Computer Society, October 2009.

[4] P. Sandvik and K. Sere. Formal Analysis and Verification of Peer-to-Peer Node Behaviour. In Antonio Liotta, Nikos Antonopoulos, Giuseppe Di Fatta, Takahiro Hara, and Quang Hieu Vu, editors, *The Third International Conference on Advances in P2P Systems (AP2PS 2011)*, pages 47–52. IARIA, November 2011.

[5] P. Sandvik, K. Sere, and M. Waldén. An Event-B Model for On-Demand Streaming. Technical Report 994, Turku Centre for Computer Science (TUCS), December 2010. `http://tucs.fi/publications/insight.php?id=tSaSeWa10a` (Accessed September 2012).

[6] A. Vlavianos, M. Iliofotou, and M. Faloutsos. BiToS: Enhancing BitTorrent for Supporting Streaming Applications. In *9th IEEE Global Internet Symposium 2006*, April 2006.