NWPT 2011

MÄLARDALEN UNIVERSITY
SWEDEN

Proceedings of the 23rd

Nordic Workshop on Programming Theory

October 26 - 28, 2011

Västerås, Sweden

Paul Pettersson and

Cristina Seceleanu (Eds.)

MRTC

Mälardalen Real-Time Research Centre
Mälardalen University
Box 883, 721 23 Västerås, Sweden

Technical report 254/2011
October 2011
ISSN 1404-3041

Preface

The objective of the Nordic Workshop on Programming Theory is to bring together researchers from (but not limited to) the Nordic and Baltic countries interested in programming theory, in order to improve mutual contacts and cooperation.

The 23*rd* Nordic Workshop on Programming Theory took place at Mälardalen University, Sweden, 26-28 October 2011. The previous workshops were held in Uppsala (1989, 1999, and 2004), Aalborg (1990), Göteborg (1991 and 1995), Bergen (1992 and 2000), Åbo (1993, 1998, 2003, and 2010), Aarhus (1994), Oslo (1996 and 2007), Tallinn (1997, 2002 and 2008), Lyngby (2001 and 2009), Copenhagen (2005), and Reykjavk (2006).

There were 36 regular presentations at the workshop. In addition, the following four invited speakers gave presentations: Werner Damm (University of Oldenburg and OFFIS, Germany), Björn Lisper (Mälardalen University, Sweden), Michael Williams (Ericsson AB, Sweden), and Glynn Winskel (University of Cambridge, United Kingdom).

The organizers would like to thank the invited speakers, the programme committee, and all the researchers who submitted papers to NWPT 2011.

Programme Committee

Michael R. Hansen	Techn. Univ. of Denmark, Denmark
Einar Broch Johnsen	Univ. of Oslo, Norway
Kim G. Larsen	Aalborg Univ., Denmark
Anna Ingólfsdóttir	Reykjavk Univ., Iceland
Bengt Nordstrom	Chalmers, Univ. of Gothenburg, Sweden
Olaf Owe	Univ. of Oslo, Norway
Paul Pettersson	Mälardalen Univ., Sweden
Gerardo Schneider	IT Univ., Chalmers — Univ. of Gothenburg, Sweden, and Univ. of Oslo, Norway
Andrei Sabelfeld	Chalmers, Univ. of Gothenburg, Sweden
Cristina Seceleanu	Mälardalen Univ., Sweden
Tarmo Uustalu	Inst. of Cubernetics, Estonia
Juri Vain	Tallinn Technical University, Estonia
Marina Waldén	Åbo Akademi Univ., Finland
Uwe Wolter	Univ. of Bergen, Norway
Wang Yi	Uppsala Univ., Sweden

Local Organizing Committee

Leo Hatvani, Ellinor Karlsson, Anna Lind, Paul Pettersson (co-chair), Malin Rosqvist, Cristina Seceleanu (co-chair), and Jagadish Suryadevara.

Table of Contents

Ensure you are wearing fireproof clothes before trying to introduce disruptive technology

Michael Williams

Ericsson AB, Sweden

Abstract

About 15 years ago, we proposed that we should use functional programming languages to implement our telecommunication switching systems. We had developed and successfully used our own functional language, Erlang to develop a small PABX. We tried to introduce this to Ericsson on a broad scale but found unexpected problems. This talk will examine to problems of introducing new disruptive software technology in an industrial context.

Accelerating Backward Reachability Analysis

Tim Strazny
Carl von Ossietzky University of Oldenburg
tim.strazny@informatik.uni-oldenburg.de

Abstract

In the context of depth-first backward reachability analysis, we identify two general operations which allow for performance improvements, while covering well-known techniques such as partial order methods and pruning. We instantiate these operations with novel *backward acceleration* techniques and employ methods of *guided search* in this context. Further, we introduce *support-based search trees*, a data structure to represent upward-closed sets (ucs's) which allows for efficient implementation of operations necessary for the analysis.

Introduction. The performance of backward coverability analysis [2] for certain classes of well-structured transition systems [5], such as reset post self-modifying Petri nets, can be vastly improved by employing heuristics—as known in directed model checking—to *guide* the search during a depth-first traversal, utilizing the system's structure. We identify general conditions for optimizing the search and introduce an instantiation of Algorithm 1 making use of *backward acceleration*, a novel means to identify and exploit recurring patterns in the search space. Furthermore, we advocate the use of *place bounds* for pruning in a Petri net context. As the efficiency of the underlying data structure plays a major role for the algorithm's performance, we investigate necessary conditions to implement equivalence classes of states, which are then represented by *support-based search trees*, a data structure we develop to provide efficient operations for the depth-first variant of the search algorithm.

```
1  W := F; V := ∅;
2  while W ≠ ∅ do
3    x := select(W); W := W \ {x};
4    if x ∈ ↓I then
5      V := min(V ∪ {x});
6      W := ∅
7    else
8      if x ∉ ↑V then
9        V := min(V ∪ {x});
10       W := min(W ∪ opt(I, V, pb(x)))
11     fi
12   fi
13 od
```

Algorithm 1: Reachability analysis.

Foundations. For a well-structured transition system $\mathscr{S} = (S, \to, \leq)$ [5], where S is a set of states, $\to \subseteq S \times S$ is a transition relation, and $\leq \subseteq S \times S$ is a well-quasi ordering, it is decidable whether a transition sequence σ starting in the downward-closure $\downarrow I := \{ y \in S \mid \exists x \in I : y \leq x \}$ of a finite set $I \subseteq S$ of initial states and ending in the upward-closure $\uparrow F := \{ y \in S \mid \exists x \in F : x \leq y \}$ of a finite set $F \subseteq S$ of final states exists, i.e. $\exists x \in \downarrow I, y \in \uparrow F : x \xrightarrow{\sigma} y$. When the variation of the backward reachability analysis [2] shown in Algorithm 1 terminates, $\uparrow V$ either forms the backward reachable state space or the loop prematurely was exited as a sequence from $\downarrow I$ to $\uparrow F$ was found. In both cases $\downarrow I \cap \uparrow V \neq \emptyset \Leftrightarrow \downarrow I \cap \mathrm{Pred}^*(\uparrow F) \neq \emptyset$ holds, where $\mathrm{Pred}^*(\uparrow F)$ represents the predecessors of the elements in $\uparrow F$ w.r.t. the reflexive, transitive closure of the transition relation. The algorithm uses the four functions

- *select*, which chooses an element from W (basis of states to process), allowing for *guided search*,
- *min*, which minimizes a finite base and depends on an efficient data structure,
- *pb*, to compute a finite basis of the set of one-step predecessors, and
- *opt*, allowing for optimization of *pb*'s output, such as *backward acceleration*.

Guided Search. The order in which elements are chosen from W does neither influence correctness nor termination of the algorithm. However, certain orderings are preferable as they lead to fewer loop iterations and in the best case—if $\uparrow F$ is reachable from $\downarrow I$—an according transition sequence is found immediately.

In the setting of Petri nets, we are able to describe a notion of *distance to* $\downarrow I$ for a state x by taking the length of the shortest path through the net's structure from a place marked in I to a place marked in x. We propose selecting one of those states from W which are closest to $\downarrow I$ as a heuristic for search guidance. Also, simple orderings such as on the number of tokens can be employed and tend to improve the overall running time, too.

Backward Acceleration. The result $O = opt(I,V,pb(x))$ of the newly introduced function *opt*, has to satisfy three conditions:

- to achieve termination, it has to be finite,
- to achieve correctness, if and only if $\downarrow I$ is backward reachable from $\uparrow pb(x)$, then $\downarrow I$ has to be backward reachable from $\uparrow O$, i.e. $\downarrow I \cap \mathrm{Pred}^*(\uparrow pb(x)) \neq \emptyset \Leftrightarrow \downarrow I \cap \mathrm{Pred}^*(\uparrow O) \neq \emptyset$, and
- to be able to present a transition sequence leading from $\downarrow I$ to $\uparrow F$ (if it exists), every element in o is backward reachable from x, i.e. $\forall y \in O\ \exists p \in pb(x) : y \in \mathrm{Pred}^*(\uparrow p)$.

These conditions leave room for well-known techniques such as *pruning* [4], *partial-order methods* [1], as well as a new path-learning method we call *backward acceleration*. Moreover, the condition for correctness is compatible with the loop invariant of Algorithm 1

$$(\mathrm{Pred}^*(\uparrow W) \cup \uparrow V) \subseteq \mathrm{Pred}^*(\uparrow F) \ \wedge\ \big(\ \downarrow I \cap \mathrm{Pred}^*(\uparrow F) \neq \emptyset \Leftrightarrow \downarrow I \cap (\mathrm{Pred}^*(\uparrow W) \cup \uparrow V) \neq \emptyset\ \big),$$

which expresses that $\uparrow V$ is backward reachable from $\uparrow F$ and $\downarrow I$ is backward reachable from $\uparrow F$ if and only if it is backward reachable from $\uparrow W$ or has a non-empty intersection with $\uparrow V$, by which partial correctness of Algorithm 1 can be shown.

During inspection of the backward search space for several case studies, certain transition sequences appeared in recurring patterns. The idea of backward acceleration is to identify such repeating patterns $x \xrightarrow{\sigma} y$, e.g. at the second occurrence, and exploit it by computing the maximal extension of the transition sequence σ, $x \xrightarrow{\tau} y'$, s.t. $\tau = \sigma^k$, k maximal. Two problems arise: On one hand, exploitable transition sequences have to be recognised—for Petri nets, the token sum may be required to have decreased in order to guarantee $y' < y$—, and on the other hand, the maximal extension of σ has to be computed. If successfully applied, backward acceleration is capable of reducing the backward reachable state space by large amounts.

As backward search algorithms suffer from traversing portions of the state space which are not (forward) reachable, pruning techniques come in handy, i.e. using over-approximations of the (forward) reachable state space to prevent states outside of the over-approximation from being explored any further. Considering the implementation of pruning by structural invariants for Petri nets [4], we decided to use a standard algorithm with a running time complexity cubic in the problem size to identify invariants, and again utilize the net's structure in order to derive more, yet weaker invariants (covering fewer places per invariant). In the extreme, we derive *bounds* on the number of tokens on individual places, while preserving more complex invariants and the relationships between places they express. Testing the violation of bounds of places is considerably faster and thus provides a good precursor to testing violation of invariants.

Support-based Search Trees. For a set $X \subseteq S$, the function *min* returns the minimal basis s.t. $\uparrow X = \uparrow \min(X)$, i.e. $\left\{ x \in X \mid \forall y \in X : y \not\leq x \right\}$. In the context of Algorithm 1, *min* is applied to the union of two sets: A basis of a ucs, V or W, and some set of states. In our implementation, adding a set of states to a ucs is done sequentially. Thus, the three primary operations to optimize for when considering a state x are (1) check if y in the ucs exists with $y \leq x$, (2) remove those elements y from the ucs which are $x \leq y$, and (3) add x to a ucs. We propose to use necessary conditions γ, s.t. $x \leq y \Rightarrow \gamma(x,y)$, to construct data structures allowing for efficient implementations of the operations to access ucs's. By intersecting several such necessary conditions, the number of comparisons to be carried out can be narrowed down drastically.

For example, when states are represented by vectors in $\mathbb{N}^d$ and $x \leq y \Leftrightarrow \forall 1 \leq i \leq d : x(i) \leq y(i)$, e.g. in Petri net settings, we define the function $supp : S \to \mathscr{P}(\mathbb{N}_{\leq d})$ mapping a state x to a set of indices s.t. $supp(x) := \{ 1 \leq i \leq d \mid x(i) \neq 0 \}$. Given two states $x, y \in \mathbb{N}^d$, $x \leq y$ implies $supp(x) \subseteq supp(y)$. Thus the support gives rise to a necessary condition for $x \leq y$. A *support-based search tree* is a binary tree where nodes are labelled with indices and sets of states, and edges are labelled with $\bot$ or $\top$, stating whether the label of the parent node belongs to the support of the states in sets attached to child nodes. This data structure allows for efficient implementations of the operations mentioned above: e.g, if $x \in \uparrow V$ is to be checked and V is stored as a support-based search tree with the root node being labelled with index 1 and $1 \notin supp(x)$, then the branch with edge labelled $\top$ can be safely ignored since no $y \in V$ with $1 \in supp(y)$ can satisfy $y \leq x$.

Furthermore, we define $\sum x$ to be the sum $\sum_{i=1}^{d} x(i)$ and again observe a necessary condition: Given two states $x, y \in \mathbb{N}^d$, $x \leq y$ implies $\sum x \leq \sum y$. By collecting sets of states of equal sum in classes and constructing a support-based search tree for each such class, a forest of support-based search trees is formed, increasing the performance further (cf. [3]). Of course, further necessary conditions may be employed.

Experiments. The methods presented here have been implemented as a part of PETRUCHIO/BW[1] which solves coverability / control-state reachability problems for reset post self-modifying Petri nets and belongs to the PETRUCHIO tool [7]. A preliminary comparison with tool MIST2/BW [4], which performs a breadth-first backward search using the interval sharing-tree data structure [6], showed that our implementation is able to solve a large number of problem instances delivered with MIST2/BW (in their .spec format) about two orders of magnitude faster. For example, repeating benchmarks for 2 hours each, the problem instance pncsacover.spec was solved in roughly 10 milliseconds average by our tool and in roughly 1 minute average by MIST2/BW. For problem instance delegatebuffer.spec, our tool took 5 minutes and MIST2/BW took over 1 day to solve it (average of three iterations). Further comparisons with other tools are to be carried out.

References.

[1] Parosh Aziz Abdulla, Bengt Jonsson, Mats Kindahl, and Doron Peled. A general approach to partial order reductions in symbolic verification. In Alan Hu and Moshe Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 379–390. Springer Berlin / Heidelberg, 1998.

[2] Parosh Aziz Abdulla, Kārlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. General decidability theorems for infinite-state systems. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, pages 313–323. IEEE Computer Society, 1996.

[3] Gianfranco Ciardo and Andrew S. Miner. Storage alternatives for large structured state spaces. In Raymond A. Marie, Brigitte Plateau, Maria Calzarossa, and Gerardo Rubino, editors, *Computer Performance Evaluation*, volume 1245 of *Lecture Notes in Computer Science*, pages 44–57. Springer Berlin / Heidelberg, 1997.

[4] Giorgio Delzanno, Jean-François Raskin, and Laurent Van Begin. Attacking symbolic state explosion. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification*, volume 2102 of *Lecture Notes in Computer Science*, pages 298–310. Springer Berlin / Heidelberg, 2001.

[5] Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256:63–92, April 2001.

[6] Pierre Ganty, Cédric Meuter, Laurent Van Begin, Gabriel Kalyon, Jean-François Raskin, and Giorgio Delzanno. Symbolic data structure for sets of k-uples of integers. Technical Report 570, Université Libre de Bruxelles, 2007.

[7] Roland Meyer and Tim Strazny. Petruchio: From dynamic networks to nets. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 175–179. Springer Berlin / Heidelberg, 2010.

[1] `http://csd.informatik.uni-oldenburg.de/~critter/petruchio.tar.gz`

From functional programming to multicore parallelism: A case study based on Presburger Arithmetic

Phan Anh Dung and Michael R. Hansen
DTU Informatics, Technical University of Denmark

The overall goal of this work is studying parallelization of functional programs with the specific case study of decision procedures for Presburger Arithmetic (PA). PA is a first order theory of integers accepting addition as its only operation. Whereas it has wide applications in different areas, we are interested in using PA in connection with the Duration Calculus Model Checker (DCMC) [5]. There are effective decision procedures for PA including Cooper's algorithm and the Omega Test; however, their complexity is extremely high with doubly exponential lower bound and triply exponential upper bound [7]. We investigate these decision procedures in the context of multicore parallelism with the hope of exploiting multicore powers. Unfortunately, we are not aware of any prior parallelism research related to decision procedures for PA. The closest work is the preliminary results on parallelism in the SMT-solver Z3 [8] which has the capability of solving Presburger formulas.

Functional programming is well-suited for the domain of decision procedures, and its immutability feature helps to reduce parallelization effort. While Haskell has progressed with a lot of parallelism-related research [6], we choose F# to be able to have explicit control over parallelism on the .NET framework and utilize its option to resort to mutation when optimizing performance.

1 Parallelization of Cooper's algorithm

The algorithm removes quantifiers in the inside-out order using the following transformation [5]:

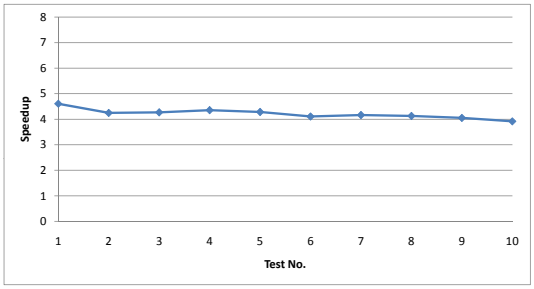
$$\exists\, \mathtt{x}.\ \phi \iff \bigvee_{\mathtt{i}=1}^{\delta} (\phi[\top/\mathtt{ax} < \mathtt{t},\ \bot/\mathtt{ax} > \mathtt{t}] \vee \bigvee_{\mathtt{ax}<\mathtt{t}} \phi[\mathtt{t}+\mathtt{i}/\mathtt{ax}] \wedge \delta' \mid \mathtt{t}+\mathtt{i}) \tag{1}$$

where a > 0, δ is the least common multiple of the coefficients of x in ϕ and δ' is the least common multiple of the divisors in divisibility constraints.

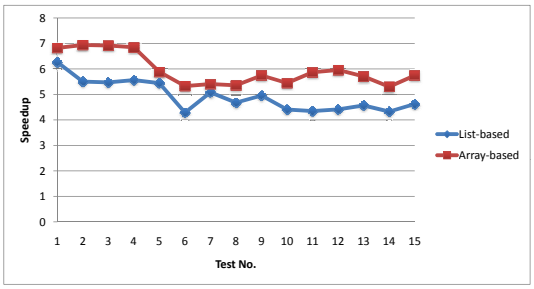
Before parallelization, several optimizations have been considered for Cooper's algorithm. Interestingly, *eliminating blocks of quantifiers* [3] has shown its good performance on the multicore platform. This procedure is superior as quantifiers are distributed into inner formulas for quantifier removal, resulting in manipulation of small data structures which is extremely fast when data is likely to fit in cache. Our preliminary test with several small formulas shows that this optimization leads to $20\times$ performance gain compared to the baseline variant [4].

Cooper's algorithm works with Presburger formulas in negation normal form (NNF). In an ideal case, many formulas need to be resolved simultaneously and we can employ parallelism constructs to utilize multicore powers efficiently. In other cases, parallelism can be extracted from the structure of Cooper's transformation. In Equation 1, denote $\mathtt{B}$ as the number of $\mathtt{ax} < \mathtt{t}$ constraints in ϕ. Despite the symbolic representation of disjunctions, $\mathtt{B}+1$ substitutions should be performed in the formula. As these substitutions are totally independent, we are able to execute them in parallel. The value of $\mathtt{B}$ increases after each elimination step; therefore, the chance of parallelism is ensured.

Degree of parallelism is even more significant if we keep Presburger formulas in disjunctive normal form (DNF). The advantage is the utilization of all available cores for parallel execution and the disadvantage is the explosion of memory usage which could go beyond the capability of the system. We have implemented this idea for parallel execution of the Omega Test where using DNF is inevitable. Detailed discussion of this procedure could be found in the next section.



(a) Cooper's algorithm



(b) The Omega Test

Figure 1: Speedup factors on an 8-core machine

2 Parallelization of the Omega Test

The *Exact Shadow* is an element of the Omega Test which is presented in the form of strict comparisons as follows [4]:

$$\exists\, \mathtt{x}.\ \beta < \mathtt{b}\mathtt{z} \wedge \mathtt{c}\mathtt{z} < \gamma \iff \mathtt{c}\beta + 1 < \mathtt{b}\gamma \tag{2}$$

The *Exact Shadow* has the similar idea to the Fourier-Motzkin elimination for real arithmetic; nevertheless, the discrete property of integer arithmetic requires either $\mathtt{b} = 1$ or $\mathtt{c} = 1$ to proceed the equivalence.

To employ this shadow, we first convert a formula into DNF and apply the shadow until the condition of coefficients is no longer satisfied. Parallelism in our algorithms is evaluated by a language-based parallel cost model, namely the *DAG model of multithreading* [2]. For example, we consider a Presburger instance in the form of a quantified formula with $\mathtt{k}$ quantifiers and a conjunction consisting of $\mathtt{m}$ literals and $\mathtt{n}$ disjunctions of two literals. Transformation of this formula into DNF results in a disjunction of $2^{\mathtt{n}}$ conjunctions of $(\mathtt{m}+\mathtt{n})$ literals. Because these conjunctions are resolved independently based on the rule $\exists\, x_1 \ldots x_n.\ \bigvee_i \bigwedge_j L_{ij} \iff \bigvee_i \exists\, x_1 \ldots x_n.\ \bigwedge_j L_{ij}$, the *parallelism factor* of this example is $\Theta(2^{\mathtt{n}})$. The algorithm contains enough parallelism; therefore, the degree of parallelism is only bounded by the number of used processors. The *DAG model of multithreading* is helpful when it allows us to predict parallel efficiency of an algorithm before proceeding further with implementation.

3 Experimental results

Due to the lack of test suites for Presburger Arithmetic, we attempt to generate some test formulas controllable in terms of size and complexity. Test formulas are formulated by using Pigeon Hole Principle as follows: *given* $\mathtt{N}$ *pigeons and* $\mathtt{K}$ *holes, if* $\mathtt{N} \leq \mathtt{K}$ *there exists a way to assign the pigeons to the holes where no hole has more than one pigeon; otherwise, no such assignment exists.*

Let $\mathtt{x_{ik}}$ be the predicate where pigeon $\mathtt{i}$ is in hole $\mathtt{k}$, and the detailed construction is described below. One important point is that functional programming allows us to express this construction in a natural way which is very close to the logical formalism.

$$\mathtt{P(N,\ K)} = \bigwedge_{\substack{1 \leq i \leq N \\ 1 \leq k \leq K}} (\mathtt{x_{ik}} \Rightarrow \bigwedge_{\substack{1 \leq j \leq N \\ j \neq i}} \neg \mathtt{x_{jk}}) \wedge \bigwedge_{1 \leq \mathtt{i} \leq \mathtt{N}} (\bigvee_{1 \leq \mathtt{j} \leq \mathtt{K}} \mathtt{x_{ij}}) \wedge \bigwedge_{\substack{1 \leq i \leq N \\ 1 \leq k \leq K}} (\mathtt{x_{ik}} \Rightarrow \bigwedge_{\substack{1 \leq j \leq K \\ j \neq k}} \neg \mathtt{x_{ij}}) \tag{3}$$

Satisfiability of Pigeon Hole formulas is known as a provably difficult case for SAT solvers [1]. We define quantified formulas with an arbitrary number of quantifiers in the form of $\exists\, \mathtt{x_{11}} \ldots \mathtt{x_{ab}}.\ \mathtt{P(N,K)}$ where $\mathtt{x_{ik}}$ is encoded as simple equalities, inequalities and divisibility constraints [4]. By doing so, we create quite challenging Presburger instances with predetermined truth values for testing purpose.

We construct the test set for Cooper's algorithm from Pigeon Hole formulas with 21-30 holes, one pigeon and 3 quantifiers for each formula. Each formula is a combination of several independent formulas, which helps to increase degree of parallelism. Results on an 8-core machine are illustrated in Figure 1a showing $4-5\times$ speedup of the parallel version compared to the corresponding sequential one. The results could be improved if one pays more attention to cache usage and minimizes the number of memory allocations.

Another group of test formulas is extracted from Presburger fragments generated by DCMC [5]. We do not describe the details of those formulas, but they have the same form as the example presented in Section 2. Moreover, they consist of many inequalities with very small coefficients (1, -1 or 0) making them become an ideal input for the *Exact Shadow* discussed above.

The test set for the Omega Test consists of formulas extracted from the model-checking problem with 5, 7 and 9 disjunctions respectively. Figure 1b shows speedup factors for the `List-based` and the `Array-based` implementations on the 8-core machine. In general, their speedups are really high with an approximate $5\times$ speedup in the worst case. One should notice that the `Array-based` variant is always more scalable than the `List-based` one. Although the number of cache misses has influence on scalability, the array-based representation is suitable for parallelization due to its advantage of keeping data close together in memory. The result here shows the advantage of using the *Exact Shadow* for alternating quantifiers. Moreover, the parallelization process is fully applicable for the Fourier-Motzkin elimination which is widely used in decision procedures for real numbers.

4 Conclusions

In this paper, we have presented our parallelism concerns regarding decision procedures for PA. A lesson learned is that cache has a huge influence on performance, and even a small change to make data fit in cache could result in $20\times$ performance gain. Moreover, multicore powers are easy to leverage using functional programming when good speedup could be obtained without much parallelization effort. We have achieved good speedups in parallelizing two decision procedures for PA, but the idea should fit the Fourier-Motzkin elimination very well. Although the results may be promising, full exploitation of multicore powers is still an open question for further research.

References

[1] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proceedings of the 39th annual Design Automation Conference*, 2002.

[2] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 1996.

[3] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[4] Phan Anh Dung. Presburger Arithmetic and its use in verification. Master's thesis, Technical University of Denmark, DTU Informatics, 2011.

[5] Michael R. Hansen and Aske W. Brekling. On Tool Support for Duration Calculus on the basis of Presburger Arithmetic. In *18th International Symposium on Temporal Representation and Reasoning*, 2011.

[6] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K Aswad, and Phil Trinder. Seq no more: Better strategies for parallel haskell. In *Haskell '10: Proceedings of the Third ACM SIGPLAN Symposium on Haskell*, 2010.

[7] Derek C. Oppen. A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger Arithmetic. *Journal of Computer and System Sciences*, 1978.

[8] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Moura. A Concurrent Portfolio Approach to SMT Solving. In *Proceedings of the 21st International Conference on Computer Aided Verification*, 2009.

GSPeeDI - a reachability checker for planar, polygonal hybrid systems using incomplete acceleration

Hallstein A. Hansen
Buskerud University College, Kongsberg, Norway
hallsteinh@hibu.no

Abstract

The Generalized Polygonal Hybrid System (GSPDI) is a class of hybrid automata that are restricted to modeling planar systems without resets, but where the reachability problem is decidable. We present the latest version of our reachability checker for GSPDIs, GSPeeDI, which implements a recently developed reachability search algorithm, and give a case study which illustrates the advantages of the new method.

1 Introduction

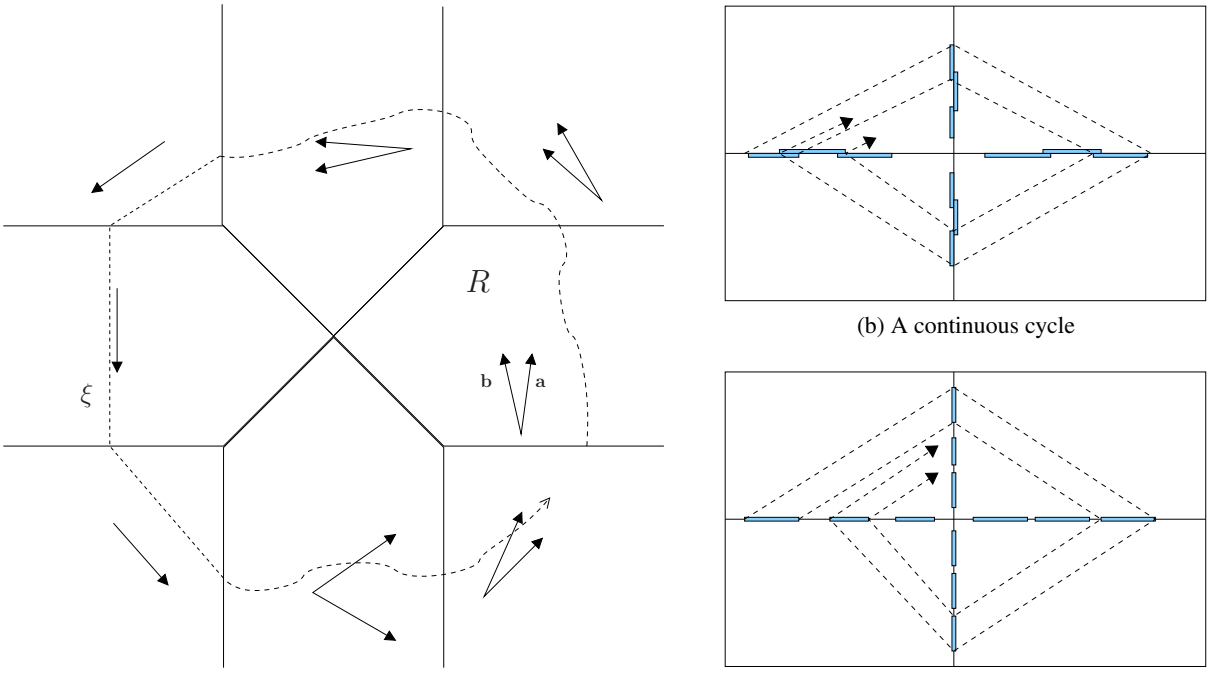
The theory of hybrid systems has been developed as a formal way of modeling and analyzing control systems [1]. Hybrid automata allow for modeling very expressive systems, as the system state can evolve following non-linear differential inclusions and change through discrete jumps. On the other hand, most properties of hybrid automata are undecidable and for automata with non-linear evolution there does not even exist an efficient algorithm for answering reachability questions [8].

The Generalized Polygonal Hybrid System (GSPDI) is a class of hybrid automata where the reachability problem is decidable [10]. GSPDIs are restricted to modeling planar systems without resets, but many real-world systems can be over-approximated by a GSPDI, such as systems controlled by a proportional controller [4]. Algorithms for solving the reachability problem have all been forced to handle iteration of cycles in the reachability graph [2, 9], but recently we have introduced a method, incomplete acceleration, that removes the need to iterate any cycle [5].

In this paper we present an updated version of GSPeeDI, a GSPDI reachability checker [6, 3], with an algorithm that implements the new method. Through a system based on the van der Pol equations we show the savings in execution time gained from not having to iterate cycles.

2 Generalized Polygonal Hybrid Systems

A Generalized Polygonal Hybrid System (GSPDI), Figure 1al, is defined on the plane and partitioned into a set of regions, separated by edges. Note that in a GSPDI we consider edge-to-edge reachability. The possible evolution of the state of a GSPDI is constrained, region-wise, by an affine differential inclusion. In Figure 1a we see a region R with its differential inclusion illustrated by the angle $\angle_{\mathbf{a}}^{\mathbf{b}}$. We also see a trajectory ξ, a particular evolution of the GSPDI in question, which is traversing the edges of the GSPDI in a *cycle* [2]. In general, a GSPDI may contain many such (simple) cycles, and a naive implementation of a reachability checker would both have to generate and iterate all cycles, which can be computationally expensive [9, 7]. Previous work has dealt with developing techniques to *accelerate* cycles instead [2, 7], by immediately computing the set reachable by any number of iterations, but this work has been restricted to continuous cycles where the reachable set grows from some initial interval (see Figure 1b). Recently we had introduced *incomplete acceleration*, a method for analyzing cycles whose iterations create a sequence of disjoint intervals [5] (see Figure 1c). Incomplete acceleration enables us to determine whether any point on the edges comprising the cycle is reachable or not, and to compute whether and where trajectories leave the cycle to another part of the GSPDI. This is all done without actually computing the reachable set on the cycle, hence the term 'incomplete'.



(a) A GSPDI

(b) A continuous cycle

(c) A disjoint cycle

Figure 1: GSPDI concepts

3 The GSPeeDI tool

The tool GSPeeDI solves the reachability question for GSPDIs, and we have updated this tool (version 2.2) with an implementation of incomplete acceleration. GSPeeDI implements a tool chain of three separate stages:

- *System to GSPDI*: A system with possibly non-linear dynamics is approximated by a GSPDI. The current version of GSPeeDI non-conservatively approximates non-linear autonomous systems.
- *GSPDI to edge-graph*: A graph of the region edges is built from a GSPDI.
- *Reachability search*: Given a GSPDI, a starting point and final point, the tool decides whether the final point is reachable from the initial point for the given GSPDI.

4 Case study: The van der Pol equation

The van der Pol oscillator is used in electrical engineering, neurology, and seismology. It is described by the following equations:

$$x'(t) = y(t) \qquad\qquad y'(t) = -1.5(x(t)-1)y(t) - x(t).$$

The van der Pol oscillator contains a limit cycle, and so GSPDIs generated from such systems give us, as shown in the screen shot of Figure 2, the opportunity to demonstrate the cycle handling capabilities of GSPeeDI. In table 1 we give some experimental results showing the difference in reachability search execution time between version 2.1.0 of GSPeeDI, which required the computation of all cycles in the GSPDI, and version 2.2.0, where cycles are accelerated when they first occur during the search. As can

GSPDI #	Number of regions	Time old version (2.1.0)	Time new version (2.2.0)	Note
1	378	50s	6s	
2	789	316s	45s	Figure 2

Table 1: Difference in reach set computation running time between GSPeeDI versions 2.1.0 and 2.2.0

be seen the old version, which used several ad-hoc optimizations to reduce running time, is more than seven times slower than the new version.

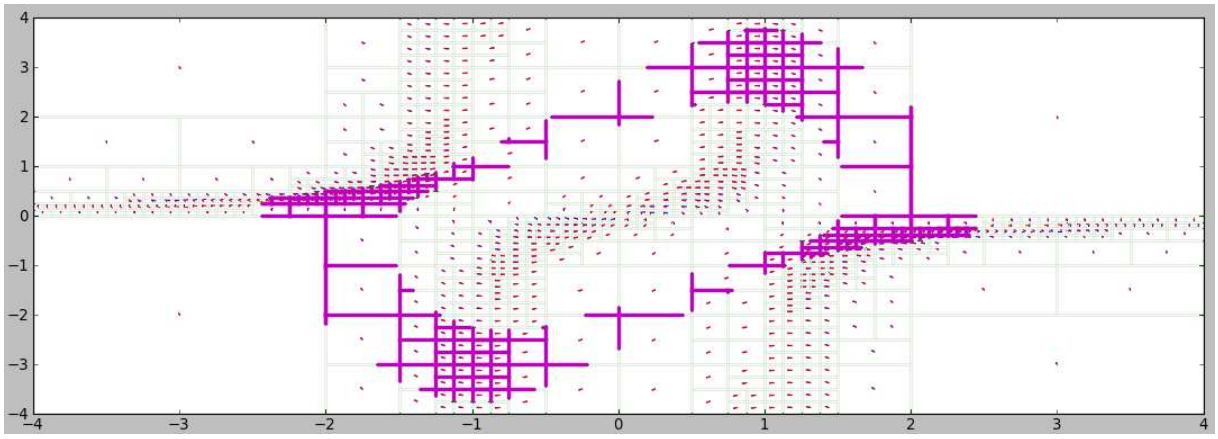


Figure 2: GSPeeDI screen shot: A GSPDI approximating the van der Pol oscillator. Reach set in violet.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computuer Science*, 138:3–34, February 1995.

[2] Eugene Asarin, Gerardo Schneider, and Sergio Yovine. Algorithmic analysis of polygonal hybrid systems, part I: Reachability. *TCS*, 379(1-2):231–265, 2007.

[3] Hallstein A. Hansen. GSPeeDI. `http://heim.ifi.uio.no/hallstah/gspeedi/`.

[4] Hallstein A. Hansen. Safety verification of non-linear, planar control systems with differential inclusions. In *8th IEEE International Conference on Embedded Software and Systems (IEEE ICESS-11)*, Changsha, China, 16-18 November 2011. IEEE Computer Society. To appear.

[5] Hallstein A. Hansen and Gerardo Schneider. Reachability analysis of complex planar hybrid systems. Unpublished.

[6] Hallstein A. Hansen and Gerardo Schneider. GSPeeDI –A Tool for Analyzing Generalized Polygonal Hybrid Systems. In *ICTAC'09*, volume 5684 of *LNCS*, pages 336–342, August 2009.

[7] Hallstein A. Hansen and Gerardo Schneider. Reachability Analysis of GSPDIs: Theory, Optimization, and Implementation. In *25th Annual ACM Symposium on Applied Computing –Software Verification and Testing track (SAC-SVT'10)*, pages 2511–2516, Sierre, Switzerland, March 22-26 2010. ACM.

[8] T.A. Henzinger, Pei-Hsin Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *Automatic Control, IEEE Transactions on*, 43(4):540 –554, April 1998.

[9] G. J. Pace. A new breadth first search algorithm for deciding spdi reachability. Technical Report CSAI2003-01, Department of Computer Science & AI, University of Malta, 2003.

[10] Gordon J. Pace and Gerardo Schneider. Relaxing goodness is still good. In *ICTAC'08*, volume 5160 of *LNCS*, pages 274–289, 2008.

Compositionality with Strong Assumptions

Hardi Hungar*
OFFIS – Institute for Information Technology
Oldenburg, Germany
hungar@offis.de

1 Overview

To develop a complex system which involves information processing, perhaps embedded into a physical enviroment, is a nontrivial task, which needs to be structured into several steps. The design will evolve, getting more precise as detail is added, and will be distributed into interacting parts, which themselves will be further elaborated until a working implementation is generated as the final result. For this implementation to meet the requirements, it is important to have clear notions of *refinement* and *decomposition*. Refinement means that a unit in design gets described with more accuracy. Sometimes, this just consists in adding detail. In other cases an abstract or conceptual point of view gets replaced by one which is closer to its technical realization so that the change is of a more fundamental nature. Decomposition introduces a level of interconnected subsystems whose collaboration shall provide all that is required of the decomposed unit. These parts—if they are already available components used in a new context—then become objects of design activities themselves and get refined or decomposed. And of course the activities on the parts should be largely independent of each other, to simplify work by limiting the required focus of attention, and to be able to perform development in a distributed fashion. And when a part is replaced by another meeting the part's original specification, this change should not affect the correct functioning of the composed system.

This sketches the landscape of what we call *compositional design*. The central notion is that of a *component*—every design entity at each stage of development, from the full system down to an atomic part of the final implementation—is called by that name. Design consists in creating components, describing them and relating them to previously designed ones. Means of descriptions may be manyfold. In this work, we consider explicitly two forms, operational ones in the form of *behaviors*, provided e. g. by models, and declarative ones, given as *specifications*. These two forms are related by explaining their meaning in a common semantical domain of traces.

The trace semantics permits to directly relate behaviors and specifications: If all traces of the behavior of a component adhere to its specification, the component is correct. Similarly, refinement is explained as trace inclusion—the refined component should be described more precisely, with less ambiguity. When the level of abstraction changes significantly, or the viewpoint gets more technical, refinement must be generalized to a relation of *realization*, where data types get changed, or structures permuted, or communication concepts replaced. It is shown that if sufficiently good specifications of the new components are available, all these design steps can be checked on the level of specifications, i. e. enabling *virtual integration*.

A distinguishing feature of the approach exposed in this work is that behaviors as well as specifications consist of two parts, a (so-called strong) *assumption*, which spells out requirements on the usage context of the component, and a *promise*, which describes how the component behaves if the strong assumption is met. The intended meaning of the strong assumption is to define conditions under which the modeling of the component is reliable. If the occurrence of phenomena not existing on a certain level of

*The work has been funded by the German Federal Ministry of Education and Research (BMBF) as part of the project "Software Plattform Embedded Systems 2020", Grant No. O1|S08045 W. The responsibility for the content rests with the author.

abstraction or not represented in the world of a particular model can be excluded by giving certain extra conditions on available observables, collecting them in a strong assumption provides a way to simplify descriptions without sacrificing accuracy. The notions of refinement and realization take this specific nature of strong assumptions into account, and the relation between the strong assumption of a composition and those of its parts is characterized in a formal way.

Many of the concepts and ideas presented here have their root in the results of the SPEEDS project [3] (Speculative and Exploratory Design in Systems Engineering, EU, 6th Framework), and draw on classic research on compositionality, see e. g. the overview in [2], as well as more recent ones [1]. Current activities, to which this work contributes, are performed within the projects CESAR (Cost-Efficient methods and processes for SAfety Relevant embedded systems, ARTEMIS JU) and SPES 2020 (Software Platform Embedded Systems 2020, Federal Ministry of Education and Research, Germany).

2 Main Definitions and Results

Definition 1 (Variables, Formulas, Specifications)**.**

1. *$\mathscr{X}$ is a set of* variables*, $x, y \in \mathscr{X}$. The set of* ports *$\mathscr{P}$ is a subset of $\mathscr{X}$. An* interface *I is a finite set of ports.*
2. *$\mathscr{F}$ is the set of formulas.*
3. *A* specification *is a pair of formulas $H = (A, F)$ with* strong assumption *A and promise F.*

Definition 2 (Implementation, Behavior)**.**

1. *The set of* implementations *$\mathscr{M}(I)$ are atomic operational descriptions, e. g. models (for the interface I).*
2. *A* behavior *is a pair $N = (A, M)$ of a* strong assumption *A and an implementation M. The strong assumption may be a formula from $\mathscr{F}(I)$ or again an implementation from $\mathscr{M}(I)$.*
3. *A component over I is a tuple $U = (I, H, B)$, where H is a specification and B is either a behavior or a* composition*.*
4. *A* composition *is $U_0 = {}_{I_0,\mathbb{L}} \|_{i=1}^{n} U_i$ is given by an* (external) interface *I_0, parts U_i with interfaces I_i and* connectors *$\mathbb{L}$ linking ports of the parts and the external interface.*

For ease of exposition in this abbreviated presentation, we will ignore most issues related to connections and corresponding renamings and assume that connected ports have the same name.

Definition 3 (Traces, Semantical Domain, Refinement)**.**

1. *$\mathscr{T}$ is the set of* points in time*, $\mathscr{V}$ is the domain of values the variables may take.*
2. *The set of traces $\mathscr{S}$ is a subset of $[\mathscr{X} \to [\mathscr{T} \to \mathscr{V}]]$.*
3. *$\mathscr{S}^2 =_{def} \mathfrak{P}(\mathscr{S}) \times \mathfrak{P}(\mathscr{S})$ is the set of* assume-promise pairs *(AP pairs).*
4. *$[\![(\Sigma, \Upsilon)]\!] =_{def} \Sigma^C \cup \Upsilon$ is the* trace extract *of an AP pair.*
5. *$(\Theta, \Xi) \in \mathscr{S}^2$* refines *($\preceq$) (Σ, Υ) if $[\![(\Theta, \Xi)]\!] \subseteq [\![(\Sigma, \Upsilon)]\!]$ and $\Theta \supseteq \Sigma$.*

6. *The parallel composition of a set of AP pairs is defined as:*

$$\|_{i=1}^{n}(\Sigma_i,\Upsilon_i) =_{def} \left(\left(\bigcap_i \Sigma_i\right) \cup \bigcup_i (\Sigma_i \cap \Upsilon_i^C), \bigcap_i \Upsilon_i\right)$$

The first component of an AP pair is the strong assumption, and the trace extract interprets a pair like an implication: Either the assumption is violated, or the promise is kept. The first condition in the definition of refinement is the common reduction on the set of possible traces (becoming more constrained). The second condition reflects the additional role of the strong assumption as defining the scope where the description is reliable: A refining entity must be *at least as reliable* as the refined one. Similar consideration guide the definition of the semantics of the parallel composition.

Definition 4 (Semantics)**.**

1. *The semantics of implementations and formulas are sets of traces* $[\![M]\!]$ *and* $[\![F]\!]$.
2. *The semantics* $[\![(Z_1,Z_2)]\!]$ *of a specification* $Z=(A,F)$ *or a behavior* $Z=(A,M)$ *is the AP pair* $([\![Z_1]\!],[\![Z_2]\!])$.
3. *The semantics of a composition is given by* $[\![\|_{i=1}^{n} U_i]\!] =_{def} \|_{i=1}^{n}([\![U_i]\!])$
4. *A component* $U=(I,H,B)$ *is* correct *if* $[\![B]\!] \precsim [\![H]\!]$.

Proposition 5 (Main Properties)**.**

1. *Refinement is a* preorder*, i. e. reflexive and transitive.*
2. *Parallel composition is monotonic w. r. t.. the refinement preorder, i. e.* $U_i \precsim V_i \;\Rightarrow\; \left(\|_{i=1}^{n} U_i\right) \precsim \left(\|_{i=1}^{n} V_i\right)$.
3. *If the parts of a composition are correct, and the main specification follows from the composition of their specifications, the composition is correct.* $U_i \precsim H_i \wedge \left(\|_{i=1}^{n} H_i\right) \precsim H \;\Rightarrow\; \left(\|_{i=1}^{n} U_i\right) \precsim H$

Item 3. is the virtual integration mentioned in the overview section, which enables distributed development. It suffices to ensure that the final implementations of the parts meet their specification for the full design to be correct. This relies mainly on the monotonicity of the parallel operator w. r. t. refinement (Item 2.).

The result can further be generalized to compositional *realizations* crossing abstraction levels, which may involve changing data types or communication paradigms or connection structures. For that, we can use more general forms of refinements, *refinement* ($\precsim_\rho$) *w. r. t. a representation function* ρ from the abstract trace domain to a concrete one, or ${}_\alpha\precsim$, which employs an *abstraction* function α in the other direction.

References

[1] Albert Benveniste, Benoît Caillaud, and Roberto Passerone. Multi-viewpoint state machines for rich component models. In Pieter Mosterman and Gabriela Nicolescu, editors, *Model-Based Design of Heterogeneous Embedded Systems*. CRC Press, 2009.

[2] Willem-Paul de Roever. The need for compositional proof systems: A survey. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, (Int. Symp. COMPOS 97)*, volume LNCS 1537, pages 1–22. Springer, 1997.

[3] SPEEDS. SPEEDS core meta-model syntax and draft semantics, 2007. SPEEDS D.2.1.c.

Towards a Programming Language for Declarative Event-based Context-sensitive Reactive Services

Søren Debois Thomas Hildebrandt Raghava Rao Mukkamala Francesco Zanitti
*{debois, hilde, rao, frza}@itu.dk
IT University of Copenhagen
Programming, Logic and Semantics Group
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark

Abstract

We present ongoing work on a new declarative and purely event-based programming language, tentatively named *DECoReS*, for *Declarative Event-based Context-sensitive Reactive Services*. The language is based on an extension of the recently developed declarative Dynamic Condition Response (DCR) Graphs model for concurrent processes, which generalizes the classical model of prime event structures to a systems model in which infinite behavior and progress constraints can be represented by finite structures. To give semantics for the DECoReS programming language the DCR Graph model is extended with parametrized events, auto events, sub processes, time and exception handling.

1 Introduction

The Dynamic Condition Response (DCR) Graph model has been developed as part of the CosmoBiz [1] and TrustCare [2] research projects with the goal to provide a formal foundation for adaptable and flexible pervasive workflow processes and services as found e.g. within the healthcare domain.

The development of the DCR Graph model takes its outset in the declarative Process Matrix workflow model [7, 8] implemented by our industrial partner Resultmaker and generalizes the classical event structure model for concurrency [10, 11] to allow for *finite* descriptions of infinite behavior (also referred to as a systems model [9]) and specification of progress constraints.

The key elements of a DCR Graph is a set of (labelled) events (e.g. representing executions of human activities in a workflow or service requests and responses), two dual relations between events referred to as the *condition* and *response* relations respectively, and two relations for *dynamically* including and excluding events from the process.

An unconstrained event can happen at any time and any number of times as long as it is included in the process. The included conditions of an event $\in$ are the events that must happen at some point before event $\in$ happens. Dually, the responses of an event $\in$ are the events that must happen at some point after event $\in$ happens, or infinitely often become excluded. Despite its simplicity, the DCR Graph model can express all ω-regular languages and thus in particular all processes that can be specified in Linear-time Temporal Logic (LTL).

Moreover, the model allows for an intuitive operational semantics and effective execution expressed by a notion of markings of the graphs. A marking is given by three sets of events $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In})$, where Ex is the set of previously executed events, Re the activities required to be executed in the future (i.e. pending responses), and In is the currently included activities.

*Authors listed alphabetically. This research is supported by the Danish Research Agency through a Knowledge Voucher granted to Exformatics (grant #10-087067, www.exformatics.com), the Trustworthy Pervasive Healthcare Services project (grant #2106-07-0019, www.trustcare.eu) and the Computer Supported Mobile Adaptive Business Processes project (grant #274-06-0415, www.cosmobiz.dk). This work funded in part by the Danish Research Agency (grant no.: 2106-080046) and the IT University of Copenhagen (the Jingling Genies projects).

The DCR Graph model has been applied to healthcare and cross-organizational case management processes identified in case studies carried out jointly with industrial partners [3,4]. The case studies led to extensions of the core model to allow nested events and a new relation between events describing a *milestone* relation [5]. An event `e` is not enabled for execution if any of its included milestones is in the set Re of pending responses. The case studies also revealed the need for distributed implementations, which led to the development of a general technique for distributing a DCR Graph based on a notion of projections [6].

Below we exemplify current work on defining and implementing a new declarative and purely event-based language based on DCR Graphs, tentatively named *DECoReS*, for Event-based Context-sensitive Reactive Services. To support the formal semantics of DECoReS, we propose extending the model of DCR Graphs with *parametrized* events, sub processes, *automatic* events, time and exception handling.

The languages and extensions are illustrated by the following example workflow process adapted from [7, 3]. The process consists of five events: The prescription of medicine and signing of the prescription by a doctor (represented by the events `prescribe` and `sign` respectively), a nurse giving the medicine to the patient (represented by the event `give`, and the nurse indicating that he does not trust the prescription (represented by the event `distrust`) and the doctor removing the prescription, represented by the event `remove`.

```
treatment process{
  doctor may prescribe<$id, $med, $qty> {
    response: administer<$id,$med,$qty>
  }
  administer<$id, $med, $qty> process
  {
    doctor must sign { exclude: remove }
    nurse must give {
      condition: Executed(sign) &
                 not Executed(remove) &
                 not Response(sign)
      exclude: sign, give, distrust, remove
    }
    nurse may distrust {
      response: sign
      include: remove
      exclude: give
    }
    doctor may remove {
      exclude: sign,give,distrust,remove
    }
  }
}
```

To capture that every prescription event `prescribe` leads to the possible execution of a "fresh" set of events `sign`, `give`, `distrust` and `remove`, we propose as the first extension of DCR Graphs to allow *sub process events* that instantiate a DCR Graph as a sub process when they are required as a response. This allows us to group the four events inside the sub process event `administer`.

The result is a process model which is more expressive than ω regular languages but still have an intuitive operational semantics as dynamically unfolding graphs.

A third proposed extension is to allow events to be called automatically. Such events are called "auto-events" and they can be emitted in order to communicate to the external world some important state of the process.

Finally, we work on adding time deadlines to the constraints, i.e. making it possible to specify that a given response must happen within a given time interval. This then again leads to the need for handling violations of such constraints. The modified process below illustrates how time constraints and exception handling can be added to the language, while also giving a small example of how the language can be used for implementing context-sensitive services.

```
doorsensor process {
  door may open {
    response: close within 15s throw door_left_open
  }
  door may close
  door_left_open process {auto leftopen}
}
```

In the above sample process, we model a process that senses when a door is being opened and closed. This is conveniently written using the role `door` for sensor at the door. The timing constraint `within 15s` specifies that in response to an `open` event, a `close` one must follow within 15 seconds, otherwise an instance of the sub process `door_left_open` will be created, which executes an autoevent `leftopen`.

Informally, an event that is required as a `response` can be annotated with a timing constraint interval `in Start within End` must happen after `Start` time units but before `End` time units; omitting the `Start` constraint is a shortcut for `in 0 within End`. Dually, a timed `condition` event, annotated with a time constraint `since Start within End`, must have happened at least `Start` time units before and at most `End` time units before. Omitting the `Start` constraint is a shortcut for `since 0 within End`.

Lastly, timed responses can be annotated with an `throw Event` construct, which requires `Event` to happen as a response, if the response do not happen within the required interval. Similarly, conditions may throw such exception events if an event happen and the condition is not satisfied.

References

[1] Thomas Hildebrandt. Computer supported mobile adaptive business processes (CosmoBiz) research project. Webpage, 2007. `http://www.cosmobiz.org/`.

[2] Thomas Hildebrandt. Trustworthy pervasive healthcare processes (TrustCare) research project. Webpage, 2008. `http://www.trustcare.dk/`.

[3] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Declarative modelling and safe distribution of healthcare workflows. In *International Symposium on Foundations of Health Information Engineering and Systems*, Johannesburg, South Africa, August 2011.

[4] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Designing a cross-organizational case management system using dynamic condition response graphs. In *Proceedings of IEEE International EDOC Conference*, 2011. to appear.

[5] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Nested dynamic condition response graphs. In *Proceedings of Fundamentals of Software Engineering (FSEN)*, April 2011. to appear.

[6] Thomas Hildebrandt, Raghava Rao Mukkamala, and Tijs Slaats. Safe distribution of declarative processes. In *9th International Conference on Software Engineering and Formal Methods (SEFM) 2011*, 2011. to appear.

[7] Karen Marie Lyng, Thomas Hildebrandt, and Raghava Rao Mukkamala. From paper based clinical practice guidelines to declarative workflow management. In *Proceedings of 2nd International Workshop on Process-oriented information systems in healthcare (ProHealth 08)*, pages 36–43, Milan, Italy, 2008. BPM 2008 Workshops.

[8] Raghava Rao Mukkamala, Thomas Hildebrandt, and Janus Boris Tøth. The resultmaker online consultant: From declarative workflow management in practice to LTL. In *Proceeding of DDBP*, 2008.

[9] Vladimiro Sassone, Mogens Nielsen, and Glynn Winskel. A classification of models for concurrency. In *Proceedings of CONCUR'93*, volume 715 of *LNCS*, 1993.

[10] Glynn Winskel. Event structures. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets*, volume 255 of *Lecture Notes in Computer Science*, pages 325–392. Springer, 1986.

[11] Glynn Winskel and Mogens Nielsen. Models for concurrency. pages 1–148, 1995.

Verified Design Patterns for Long-running Nested Transactions

Saleem Vighio
Department of Computer Science
Aalborg University
Aalborg, Denmark
vighio@cs.aau.dk

Anders P. Ravn
Department of Computer Science
Aalborg University
Aalborg, Denmark
apr@cs.aau.dk

Zhiming Liu
International Institute for Software Technology
United Nations University
Macau SAR, China
lzm@iist.unu.edu

Abstract

We present formal analysis of web services patterns which support compensation in complex distributed business processes. The patterns separate business logic including compensating actions from coordination protocols. The patterns are illustrated by a version of the well known travel reservation process example which is extended with a nested transaction. Our analysis is performed with the model checking tool UPPAAL. and it is shown that the model satisfies key lproperties.

1 Introduction

Transactions are a very important concept for SOA-based applications for the use by enterprises interacting beyond their organizational boundaries. The transactions may run for a longer time and here the Web Services Business Activity (WS-BA) standard [4], support long-running business transactions and address cross-organizational communication. Yet, it is far from trivial to implement the protocols correctly, for instance i[10] analyzes use of the WS-BA standard, and a key finding is that, compensations should be handled at the WS-BA standard level, rather than in application business logic. Thus they propose a standardized middleware. Also in [9] a transactional middleware is introduced to separate coordination logic from business logic, for easier client implementation. This approach is further extended in [2], to uncouple initiator and coordinator and thus suggests possible extensions to the standard. We investigate an alternative light-weight solution, where the business logic is embedded in compensation protocol patterns. Thus we achieve the same goal without proposing any changes to the standard or introduction of new standardized middleware. Another important property of the WS-BA standard , is its ability to deal with nested transactions [3]. Thus, a nested transaction is an extension of a simple/flat transaction, whose operations execute sequentially. Nested transactions can have a hierarchical tree-like structure, where every leaf is a simple transaction and every intermediate node is a nested transaction. This adds to the power of the concepts, but requires some care in defining the pattern.

The service implementation patterns, proposed here, are complex, thus we want to ensure that they are correct. This we do by modelling an instance of them in the model checker UPPAAL [1] and checking key proerties, for instance consistent termination.

2 Patterns and Verfication

The patterns are illustrated by the well-known travel reservation process. Initially, a *Client* sends a reservation request for an air-ticket and a hotel room to the *TravelAgent*. The *TravelAgent* in turn requests a room from the *Hotel* and a seat from the *Airline*. We enhance the example with a nested transaction so

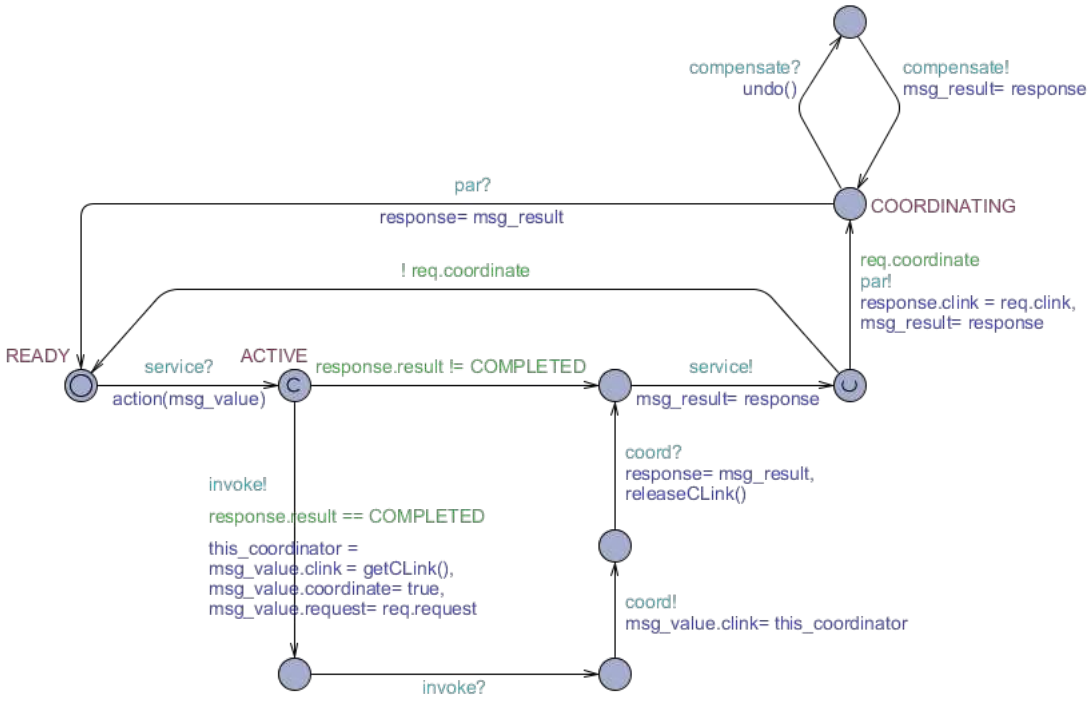


Figure 1: Complex service

that the *Airline* after receiving a request from the *TravelAgent* requests a room and a car for the *Hotel* and the *RentACar* (car-rental-agency) respectively, for the pilot. After the *Airline* has coordinated responses from the *Hotel* and the *RentACar*, it sends a response to the *TravelAgent*. Finally, the *TravelAgent* after coordinating responses from the *Airline* and the *Hotel* sends a decision to the *Client*.

Client The UPPAAL model of the *Client* process is simple. Initially, it sends a service request on a channel `service!`. A variable contains the arguments for the message: a `ClientRequest` specifies the kind of request. a `CoLinkId` identifies a potential coordinator for services, and `coordinate` indicates whether coordination is required.

It is worth mentioning that the communication between processes follows a standard rendezvous pattern, so these interactions may be replaced by procedure calls when appropriate in an implementation.

Complex Service Services may be simple, without coordination, or complex. The model is shown in Figure 1. a simple one omits coordination. Once the process receives a service request (`service?`), it executes simple business logic, `action(msg_value)`. if the simple logic succeeds, `response.result = = COMPLETED`, the service invokes (`invoke!`) a more complex business logic which requires coordination with other services , thus it gets and sets the coordination link. When the logic returns, the service invokes (`coord!`) the coordination protocol. The coordination ends with release of a coordination link, `releaseCLink()`. Now the service has to check whether it has to participate in coordination with a higher level service with its (`par!`) participant protocol. In the location `COORDINATING`, the complex service can be told (`compensate?`), to perform a compensation. If nothing goes wrong at the participant protocol then the complex service is informed of the successful completion of the service.

Participant and Coordination Protocols Services communicate using the coordination and participant protocols. We refer the reader to [4] [7] for an in-depth description of the protocols. The coordina-

tion protocol we implement is BAwPC and the coordination logic we implement is AtomicOutcome.

Verification We formulate properties using the UPPAAL query language (a subset of TCTL). The first property we verify is termination. All processes (*TravelAgent*, *Airline*, *Hotel* and *RentACar*) and their corresponding business logic and protocols finish in their final idle state. Furthermore, we verify by simulation that in a case when one of the transactions fails, its corresponding compensation is performed. Finally, we verify the consistency on the outcome of the transaction by ensuring that if all the parties finish their work without failing then the overall outcome of the transaction is positive(`COMPLETED`).

3 Conclusion and Future Work

In this paper, we provided formal verification of service patterns based on the well-known example of a travel reservation process. The main contributions of our work arethe separation of business logic from coordination protocols and implementation of nested transactions.

One immediate extension of this work is to present the pattern in WS-BPEL notatios so it is accessible to application programmers. A more intricate point is to be able to detect loops in transaction nesting, because a loop will destroy nesting. However this is beyond static model checking.

References

[1] G. Behrmann, A. David, and K.G. Larsen. A tutorial on UPPAAL. In *Proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04)*, number 3185 in LNCS, pages 200–236. Springer-Verlag, 2004.

[2] Hannes Erven, Georg Hicker, Christian Huemer, and Marco Zaptletal. The web services-businessactivity-initiator (ws-ba-i) protocol: an extension to the web services-businessactivity specification. *IEEE International Conference on Web Services*, 0:216–224, 2007.

[3] J. E.B. Moss. *Nested transactions: an approach to reliable distributed computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[4] E. Newcomer and I. Robinson (chairs). Web services business activity (WS-businessactivity) version 1.0, 2005. http://specs.xmlsoap.org/ws/2004/10/wsba/wsba.pdf.

[5] E. Newcomer and I. Robinson (chairs). Web services atomic transaction (WS-atomic transaction) version 1.2, 2009. http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html.

[6] E. Newcomer and I. Robinson (chairs). Web services coordination (WS-coordination) version 1.2, 2009. http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html.

[7] Anders P. Ravn, Jiří Srba, and Saleem Vighio. Modelling and verification of web services business activity protocol. In *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, TACAS'11/ETAPS'11, pages 357–371, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] S. Vighio, A.P. Ravn, and Z. Liu. UPPAAL model of the travel reservation process, 2011. http://people.cs.aau.dk/∼vighio/nwpt2011.zip.

[9] Friedrich H. Vogt, Simon Zambrovski, Boris Gruschko, Peter Furniss, and Alastair Green. Implementing web service protocols in soa: Ws-coordination and ws-businessactivity. In *Proceedings of the Seventh IEEE International Conference on E-Commerce Technology Workshops*, pages 21–28, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Frederic Wenzel, Patrick Freudenstein, and Martin Nussbaumer. Strengths and weaknesses of ws-businessactivity for cross-organizational soa applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, pages 42–49, Washington, DC, USA, 2009. IEEE Computer Society.

Formal Modelling of Inter-Peer Relations in Peer-to-Peer Media Distribution Systems

Luigia Petre[1] and Petter Sandvik[1,2]
[1]Department of Information Technologies, Åbo Akademi University
[2]Turku Centre for Computer Science (TUCS)
Joukahaisenkatu 3–5, 20520 Turku, Finland
{luigia.petre,petter.sandvik}@abo.fi

1 Introduction

In recent years, there has been a trend of moving away from the traditional client-server model in network software towards peer-to-peer networks and other many-to-many relations. Especially when it comes to large scale content transfer, peer-to-peer applications and protocols such as BitTorrent [7] have become popular [18], and even found their way into electronic appliances such as network routers [4] and television sets [20]. In short, the paradigm switch from client-server communication models to BitTorrent-supporting networks amounts to enabling "clients" that are already downloading e.g., video streams, to also become "servers" for other potential clients that may download the same content. The participation of every peer in content communication provides a tremendous increase in the communication efficiency, in the communication model flexibility, and in the content availability. It is therefore highly beneficial to have a thorough understanding of this communication paradigm, to uncover its potential weaknesses and recognise how to avoid them. We aim towards this goal by developing and analysing models of a peer-to-peer media distribution system, in which all the parts, from the network structure up to the content playback, have been formally modelled and verified. We have previously looked at and modelled different parts of such a system, including algorithms for acquiring pieces of media content [16, 17] and parts of a video decoding process [15], and here we expand into modelling how peers in a peer-to-peer media distribution system could discover and interact with each other, i.e., inter-peer relations.

In swarm-like peer-to-peer systems, where peers interact only when interested in the same content, a peer that is unable to receive incoming connections, for instance when it is behind a firewall, is at a serious disadvantage compared to other peers [9]. Extensions to the original BitTorrent protocol such as peer exchange (PEX) and "distributed sloppy hash table" (DHT) [14] have been developed to alleviate this problem, and we need a reusable, extendable model of peer discovery and connectivity to be able to model these. Peer-to-peer systems and other distributed architectures have been formally modelled before [12, 22, 23], but our focus here is on creating a reusable formal model of inter-peer relations using BitTorrent as our base protocol.

2 Event-B

We develop our models based on the Event-B formalism [2], which offers excellent tool support in form of the Rodin platform [3, 10]. Event-B has its roots in the B Method [1] and the Action Systems framework [5, 6, 21]. When developing models in Event-B, the primary concept is that of abstraction [2], as *models* are created from abstract specifications and then refined stepwise into more concrete implementations. By using superposition refinement [13], i.e., adding new variables and functionality to our model in a way that prevents the old functionality from being disturbed [19], we achieve a reliable system. We prove the correctness of each step of the development using the Rodin platform, which automatically generates *proof obligations*. These are mathematical formulas to prove in order to ensure correctness; the proving can be done automatically or interactively using the Rodin platform tool. The immediate feedback from the provers makes it possible to adapt our model to better suit automatic proving, and this ability to interleave modelling and proving is a big advantage of development in Event-B using the Rodin platform.

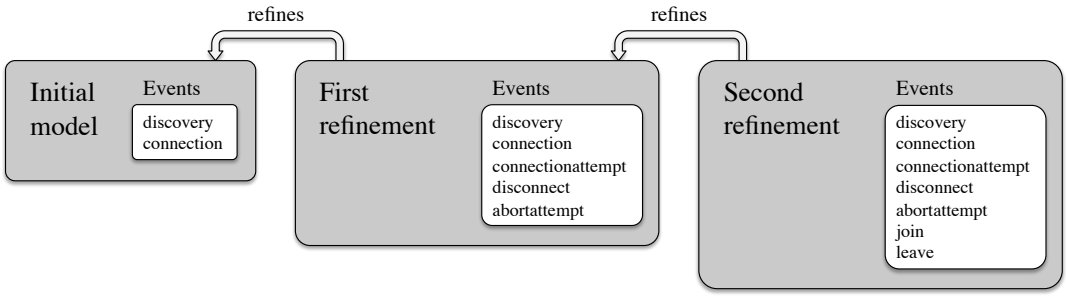


Figure 1: Model Development

3 Modelling Inter-Peer Relations

We illustrate our model development in Fig. 1. Our initial model is very abstract, with only two major functions. The first concerns one peer becoming aware of other peers. This corresponds to receiving a list of other peers from a tracker, i.e., a server that keeps track of which peers are involved in sharing a particular content, in BitTorrent. The second major function creates a connection between a peer and another peer, where the first peer must be aware of the second but not necessarily vice versa. To model these functions, we create sets of relations between peers, assuming peers are represented by natural numbers for simplicity. An "awareness" relation from 1 to 2 thereby means that peer 1 is aware of peer 2, which is different from a relation from 2 to 1. For the "connection" relation, we note that in practice we only have one connection between two peers, because in BitTorrent connections are symmetrical and traffic can flow in both directions [8]. For that reason, we allow only one connection per peer pair here, e.g., if a "connection" relation exists from 1 to 2 we do not allow one from 2 to 1. Our initial model therefore is composed of two events; *discovery*, which creates "awareness" relations from a peer to a subset of other peers, and *connection*, which creates a "connection" relation between a peer and another if there is an "awareness" relation from the first to the second.

For our first refinement step, we limit the amount of connections a peer can have, because otherwise every peer would eventually end up being connected to all other peers. While this would be possible when the number of peers is low, it would be unrealistic for a large system, and we therefore introduce a connection limit specific to each peer. This means that a connection between two peers may not always be possible, and therefore we also need to modify our connection functionality. Because peers do not know whether another peer can accept their connection or not, we replace our single connection event with two events. The *connectionattempt* event takes a peer whose connection limit has not been reached and another peer that the first peer is aware of but not connected to, and adds a "connection attempt" relation from the first peer to the second one. The *connection* event here takes a peer whose connection limit has not been reached and another peer such that there is a "connection attempt" relation from the second to the first, and creates a "connection" relation from the second to the first while removing the corresponding "connection attempt" relation. We also add two more events, *abortattempt* for aborting a connection attempt, which in practice would happen after a time limit, and *disconnect* for removing the connection between two peers, which could occur because of network issues or that the peer is no longer participating in the swarm. However, peers also close connections that have not seen any traffic for a while, and Iliofotou et al claim that the differences in download speed between BitTorrent clients can be partly attributed to differences in when they decide to do so [11]. For this reason it is important for us to model such an event that later can be refined into different types of disconnection events.

In later refinement steps, we introduce the concept of peers not being able to accept incoming connections, i.e., not being able to have "connection" relations from another peer to itself. First we achieve this in an abstract way, by simply having a boolean variable for each peer and checking the value of that variable before allowing the connection to be created. Later we refine this into a set of more complex relations, such as in the real-life situation where two peers are behind the same firewall and thereby able to accept incoming connections from each other but not from other peers. We also refine the *discovery*

event from its abstract form into more concrete ones that correspond to actual peer discovery protocols, including the DHT and PEX protocols mentioned in Section 1. Furthermore, we refine our model to include *join* and *leave* events for when peers join and leave the swarm, respectively.

Based on this complete model of connections between peers, we can further refine our model into including the message passing that later occurs between peers, thereby creating a bridge onto our previous work [17].

4 Conclusions

With the aim of modelling and analysing a whole, fully featured peer-to-peer media distribution system, we have used Event-B to model inter-peer relations in a BitTorrent-like peer-to-peer network. We have started from an abstract specification and stepwise introduced functionality until approaching a concrete implementation. Our focus has been on creating our model of such a system in a way that allows it to be reused and extended for different protocol additions, while keeping the reliability of the system intact. This gives us a foundation from which we can develop a well behaving and scalable peer-to-peer media distribution system.

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

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

[3] J.-R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[4] ASUS RT-N56U Black Diamond Combines High Speed Network Performance and Unique Design in Style. `http://www.asus.com/Networks/Wireless_Routers/RTN56U/` (Accessed September 2011).

[5] R.J.R. Back and R. Kurki-Suonio. Decentralization of Process Nets with Centralized Control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[6] R.J.R. Back and K. Sere. From Modular Systems to Action Systems. *Software - Concepts and Tools*, 13:26–39, 1996.

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

[8] B. Cohen. The BitTorrent Protocol Specification. `http://www.bittorrent.org/beps/bep_0003.html` (Accessed September 2011), January 2008.

[9] L. D'Acunto, M. Meulpolder, R. Rahman, J.A. Pouwelse, and H.J. Sips. Modeling and Analyzing the Effects of Firewalls and NATs in P2P Swarming Systems. In *IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum (IPDPSW)*, 2010.

[10] Event-B and the Rodin Platform. `http://www.event-b.org/` (Accessed September 2011).

[11] M. Iliofotou, G. Siganos, X. Yang, and P. Rodriguez. Comparing BitTorrent Clients in the Wild: The Case of Download Speed. In *USENIX IPTPS (in conjunction with NSDI 2010)*, April 2010.

[12] M. Kamali, L. Laibinis, L. Petre, and K. Sere. Self-Recovering Sensor-Actor Networks. In *FOCLASA*, 2010.

[13] S.M. Katz. A Superimposition Control Construct for Distributed Systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.

[14] A. Loewenstern. DHT Protocol. `http://www.bittorrent.org/beps/bep_0005.html` (Accessed September 2011), 2008.

[15] K. Lumme, L. Petre, P. Sandvik, and K. Sere. Towards Dependable H.264 Decoding. In *Workshop Proceedings of the Fifth IFIP WG 11.11 International Conference on Trust Management (IFIPTM 2011)*, pages 325–337, June 2011.

[16] P. Sandvik and M. Neovius. The Distance-Availability Weighted Piece Selection Method for BitTorrent: A BitTorrent Piece Selection Method for On-Demand Streaming. In *Proceedings of AP2PS '09*, October 2009.

[17] P. Sandvik and K. Sere. Formal Analysis and Verification of Peer-to-Peer Node Behaviour. In *Proceedings of AP2PS '11*, November 2011.

[18] H. Schulze and K. Mochalski. Ipoque Internet Study 2008/2009. `http://www.ipoque.com/en/resources/internet-studies` (Accessed September 2011).

[19] K. Sere. A Formalization of Superposition Refinement. In *Proceedings of the 2nd Israel Symposium on the Theory and Computing Systems*, June 1993.

[20] Vestel to Launch the First Bittorrent Certified Smart TV. `http://www.bittorrent.com/company/about/vestel_to_launch_the_first_bittorrent_certified_smart_tv` (Accessed September 2011).

[21] M. Waldén and K. Sere. Reasoning About Action Systems Using the B-Method. *Formal Methods in Systems Design*, 13:5–35, 1998.

[22] L. Yan. A Formal Architectural Model for Peer-to-Peer Systems. In X. Shen, H. Yu, J. Buford, and M. Akon, editors, *Handbook of Peer-to-Peer Networking 2010 Part 12*, pages 1295–1314. Springer US, 2010.

[23] L. Yan and J. Ni. Building a Formal Framework for Mobile Ad Hoc Computing. In *Proceedings of the International Conference on Computational Science (ICCS'04)*, June 2004.

A Functional Language for Specifying Business Reports

Patrick Bahr
Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Abstract

We describe our work on developing a functional domain specific language for specifying business reports. The report specification language is part of a novel enterprise resource planing system based on the idea of a providing a lean core system that is highly customisable via a variety of domain specific languages.

1 Introduction

Process-oriented event-driven transaction systems (POETS) is a novel software architecture for enterprise resource planning (ERP) systems, introduced by Henglein et al. [1]. Rather than storing both transactional data and implicit process state in a database, POETS employs a pragmatic separation between (a) transactional data, that is what has happened; (b) reports, that is what can be derived from the transactional data; and (c) contracts, that is which transactions are expected in the future. Moreover, rather than using general purpose programming languages to specify business processes, POETS utilises declarative domain-specific languages (DSLs) to customise the different aspects of a system. The use of DSLs not only enables explicit formalisation of business processes, it also minimises the gap between requirements and a running system.

A simplified overview over the POETS architecture is presented in Figure 1. At the heart of the system is the event log, which is an append-only list of transactions. Transactions represent relevant events that may occur, such as a payment by a customer, a delivery of goods by a shipping agency, or a movement of items into an inventory. This does not only satisfies the legal requirement for ERP systems to archive all transaction data that is relevant for auditing but also makes it possible to compute reports incrementally as shown by Nissen and Larsen [3].

2 The Report Language

The purpose of the report engine is to provide a structured view of the data base that is constituted by the system's event log. This structured view of the data in the event log comes in the form of a *report*, a collection of condensed structured information compiled from the event log. Conceptually, a report is compiled from the event log by a function of type $\mathsf{EventLog} \to \mathsf{Report}$, a *report function*. The *report language* provides a means to specify such a report function in a declarative manner.

The report language is – much like the query fragment of *SQL* – a functional language *without side effects*. It only provides operations to non-destructively manipulate and combine values. Since the system's storage is based on a shallow event log – basically a list of event representations – the report language has to provide operations to relate, filter and aggregate pieces of information. Moreover, as the data stored in the event log is inherently heterogeneous – containing data of different kind – the report language needs to offer a comprehensive type system that allows to safely operate in this setting.

The entire system is based on a common basis of base types consisting of strings, Booleans, integers, lists etc. Apart from that the system offers user-defined record types with an inheritance system based on *nominal subtyping*. To this end, POETS also offers an *ontology language* that is used to describe record types, their fields and their interdependence. The central record type is Event, which represents the events that are registered in the event log. In fact, as far as the report language is concerned, the event log is a value of type $[\mathsf{Event}]$.

Every interaction with the running system is reflected with a corresponding value of (a subtype of) type Event in the event log. The simplest example is the interface to the report engine itself. It allows to add, modify and remove reports. Each such operation is reflected by an event of type

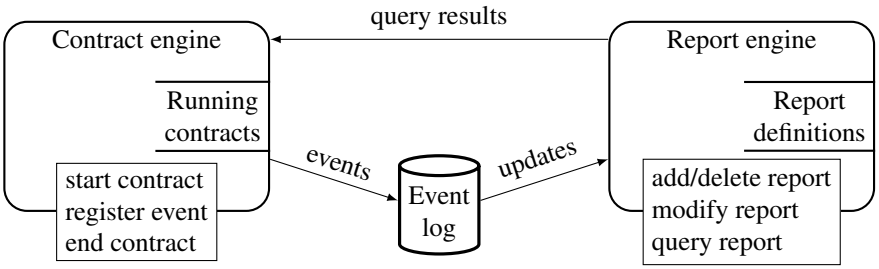


Figure 1: Bird's-eye view of the POETS architecture [1].

CreateReport, UpdateReport, and DeleteReport, respectively. The former two are subtypes of PutReport, which in turn is – like DeleteReport – a subtype of ReportEvent.

This allows us to write the following simple report function that creates the report which lists the names of all active (i.e. not deleted) reports:

```
reportNames : [String]
reportNames = [pr.name |
   cr : CreateReport ← events,
   pr : PutReport = head [ur |
         ur : ReportEvent ← events,
         ur.name ≡ cr.name]]
```

Every report function implicitly has as its first argument the event log of type [Event] – a list of events – bound to the name **events**. The syntax of the report language – and to large parts also its semantics – is based on Haskell [2]. The central data structure is that of lists. In order to formulate operations on lists concisely, we use list comprehensions [4] as seen in the above example. A list comprehension of the form $[\ e \mid c\ ]$ denotes a list containing elements of the form e generated by c, where c is a sequence of *generators* and *filters*.

As we have mentioned, access to type information and its propagation to subsequent computations is essential due to the fact that the event log is a list of heterogeneously typed elements – events of different kinds. The generator cr : CreateReport ← **events** iterates through elements of the list **events** binding each element to the variable cr. The typing cr : CreateReport restricts this iteration to elements of type CreateReport. This type information is propagated through the subsequent generators and filters of the list comprehension. In the filter $ur.name \equiv cr.name$, we use the fact that elements of type ReportEvents have a field *name* of type **String**. When binding the first element of the result of the nested list comprehension to the variable pr it is also checked whether this element is in fact of type PutReport. Thus we ignore reports that are marked as deleted via a DeleteReport event.

The report language is based on the simply typed lambda calculus extended with a polymorphic (non-recursive) let expression and a type case expression. The core language is given by the following grammar:

$$e ::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid \textbf{let}\ x = e\ \textbf{in}\ e' \\ \mid \textbf{type}\ x = e\ \textbf{of}\ \{r \to e_1; \_ \to e_2\}$$

where x ranges over variables, and c over constants which includes integers, Booleans, tuple and list constructors as well as operations on them like $+$, *if-then-else* etc. In particular, we have a fold operation **fold** of type $(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta$. This is the only operation of the report language that permits recursive computations on lists. List comprehensions are mere syntactic sugar and can be reduced to **fold** and let expressions as for example in Haskell [2].

The extended list comprehension of the report language that allow filtering according to run time type information depend on type case expressions of the form **type** $x = e$ **of** $\{r \to e_1; \_ \to e_2\}$. In such a type case expression, an expression e of some record type r_e gets evaluated to record value v which is then bound to a variable x. The record type r that the record value v is matched against can be any subtype of r_e. The further evaluation of the type case expression depends on the type r_v of the record value v. This type can be any subtype of r_e.

If $r_v \leq r$, the evaluation proceeds with e_1, otherwise with e_2. Binding e to a variable x allows to use the stricter type r in the expression e_1.

Although, the subtyping discipline that we use is nominal, the type system also allows the programmer to use record types as if the subtyping was purely structural. This is needed in order to allow the sharing of field names between distinct record types. To this end, we use *type constraints* of the form $\alpha.f : \tau$ which intuitively states that α is a record type with a field f of type τ. Field selectors are merely postfix operators. For example the *.name* field selector in the example is of type $\alpha.\mathsf{name} : \beta \Rightarrow \alpha \to \beta$.

Another important aspect of POETS in general and the report language in particular is the maintaining of references and the access of the data they refer to. This becomes necessary as certain pieces of information, e.g. customer information, are attached to a unique entity with lifecycle, e.g. a customer. To this end, POETS allow to create an entity with a unique id – a *reference*. Subsequently, information attached to this entity can be updated and eventually, the entity can be removed altogether. All these changes are, of course, reflected in the event log and can thus be examined by a report function. Nevertheless, due to the importance of references, the report language offers dedicated dereferencing operations that allow quick and typesafe access to the data associated with entities.

While the type system is important in order to avoid obvious specification errors, it is also important to ensure a fast execution of the thus obtained functional specifications. This is, of course, a general issue for querying systems. In our system, it is, however, of even greater importance since shifting the structure of the data – from the data store to the domain of queries – means that queries operate on the complete data set of the data base and thus each report has to be recomputed after each transaction. In other words, if treated naïvely, the conceptual simplification provided by the flat event log has to be paid via much more expensive computations.

This issue can be addressed by transforming a given report function f into an incremental function f' which updates a previously computed report according to the changes that have occurred since the report was computed before. That is, given an event log l and an update to it $l \oplus e$, we require that $f(l \oplus e) = f'(f(l), e)$. The new report $f(l \oplus e)$ is obtained by updating the previous report $f(l)$ according to the changes e. In the case of the event log, we have a list structure. Changes only occur *monotonically*, by adding new elements to it: Given an event log l and a new event e, the new event log is $e\#l$, where # is the list constructor of type $\alpha \to [\alpha] \to [\alpha]$.

Here it is crucial that we have restricted the report language such that operations on lists are limited to the higher-order function **fold**. The fundamental idea of incrementalising report functions isbased on the following equation:

$$\textbf{fold}\ f\ e\ (x\#xs) = f\ x\ (\textbf{fold}\ f\ e\ (xs))$$

Based on this idea, we are able to make the computation of most reports independent of the size of the event log but only dependent of the changes to the event log and the previous report [3]. Unfortunately, if we have for example list comprehensions containing more than one generator, we get report functions with nested folds. In order to properly incrementalise such functions, we need to move from list structures to multisets. This is, however, only rarely a practical restriction since most aggregation functions are based on commutative binary operations and are thus oblivious to ordering.

References

[1] Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. POETS: Process-oriented event-driven transaction systems. *Journal of Logic and Algebraic Programming*, 78(5):381–401, May 2009.

[2] Simon Marlow. *Haskell 2010 Language Report*, 2010.

[3] Michael Nissen and Ken Friis Larsen. FunSETL — Functional Reporting for ERP Systems. In Olaf Chitil, editor, *IFL '07*, pages 268–289, 2007.

[4] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(04):461–493, 1992.

Software Verification Using k-Induction*

Alastair F. Donaldson, Imperial College London
Leopold Haller, Oxford University
Daniel Kroening, Oxford University
Philipp Rümmer, Uppsala University

Abstract

We present *combined-case k-induction*, a novel technique for verifying software programs. This technique draws on the strengths of the classical inductive-invariant method and a recent application of k-induction to program verification. We present a new k-induction rule that takes an unstructured, reducible control flow graph (CFG), a natural loop occurring in the CFG, and a positive integer k, and constructs a single CFG in which the given loop is eliminated via an unwinding proportional to k. Experiments, using two implementations of the k-induction method and a large set of benchmarks, demonstrate that our k-induction technique frequently allows program verification to succeed using significantly weaker loop invariants than are required with the standard inductive invariant approach.

This abstract is based on a paper presented at the 18th Intern. Static Analysis Symposium [5].

1 Introduction

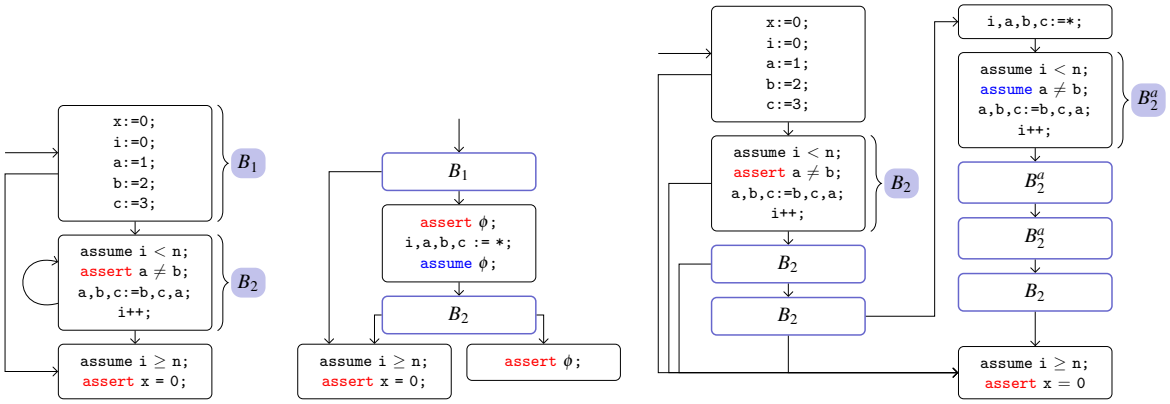
We present a novel technique for verifying imperative programs using k-induction [10]. Our method brings together two lines of existing research: the standard approach to program verification using *inductive invariants* [8], employed by practical program verifiers (including [2, 3, 4, 9], among many others) and a recent k-induction method for program verification [6, 7] which we refer to here as *split-case k-induction*. Our method, which we call *combined-case k-induction*, is directly stronger than both the inductive invariant approach and split-case k-induction.

The contributions made in this work are: 1. We formally present combined-case k-induction as a proof rule operating on control flow graphs, and state soundness of the rule; 2. We state a confluence theorem, showing that in a multi-loop program the order in which our rule is applied to loops does not affect the result of verification; 3. We present two implementations of our method: K-INDUCTOR, a verifier for C programs, and K-BOOGIE, an extension of the Boogie tool, and experimental results applying these tools to a large set of benchmarks. The experiments demonstrate that combined-case k-induction is an efficient automated verification technique for certain domains (in our case, checking the absence of DMA races in data processing programs for the Cell BE processor [6]), but is also a convenient and robust proof rule for deductive verification systems (such as Spec# [2] or Dafny [9]) that frequently allows verification to succeed with weaker loop invariants than using previous loop rules [1].

Compared with our previous work on k-induction techniques for software [6, 7], which are restricted to programs containing a single *while* loop (supporting multiple loops only via a translation of all program loops to a single, monolithic loop), our novel proof rule handles multiple natural loops in *arbitrary* reducible control-flow graphs. Furthermore, like existing loop rules for unstructured programs [1], our proof rule does not generate separate base and step cases, but combines both cases into a single CFG. This can lead to significant verification speed-ups in the presence of multiple loops, compared to split-case k-induction. The combination of base and step case into a single CFG also enables more elegant treatment of variables that are not modified in a loop, reducing the amount of auxiliary invariants that have to be provided manually or derived using techniques like abstract interpretation.

Throughout the paper, we are concerned with proving partial correctness with respect to assertions: establishing that whenever a statement *assert* ϕ is executed, the expression ϕ evaluates to true. We shall simply use *correctness* to refer to this notion of partial correctness.

*Supported by the EU FP7 STREP MOGENTES (project ID ICT-216679), the EU FP7 STREP PINCETTE (project ID ICT-257647), EPSRC projects EP/G026254/1 and EP/G051100/1, and a grant from Toyota Motors.



(a) Original CFG (b) CFG after loop cutting (c) Result of applying combined-case k-induction

Figure 1: A simple program, and verification using inductive invariants and combined-case k-induction.

2 Combined-case k-Induction by Example

We present programs as control flow graphs (CFGs) and use the terms *program* and *CFG* synonymously. We follow the standard approach of modelling control flow using a combination of nondeterministic branches and *assume* statements. During execution, a statement *assume* ϕ causes execution to silently (and non-erroneously) halt if the expression ϕ evaluates to false, and does nothing otherwise.

Consider the simple example program of Figure 1(a). The program initialises a, b and c to distinct values, and then repeatedly cycles their values, asserting that a and b never become equal. The condition for the loop is $i < n$, and is encoded using *assume* statements at the start of the loop body, and at the start of the node immediately following the loop. Variable x is initialised to zero, and after the loop an assertion checks that x has not changed. The program is clearly correct.

The inductive invariant approach. To formally prove a program's correctness using inductive invariants, one first associates a candidate invariant with each loop header in the program. One then shows that 1. the candidate invariants are indeed loop invariants, and 2. these loop invariants are strong enough to imply that no assertion in the program can fail.

A technique for performing these checks in the context of unstructured programs is detailed in [1]. The technique transforms a CFG with loops into a loop-free CFG. Each loop header in the original CFG is prepended in the transformed CFG with a basic block that: asserts the loop invariant, havocs each loop-modified variable,[1] and assumes the loop invariant. Loop entry edges in the original CFG are replaced with edges to these new blocks in the transformed CFG. Each back edge in the original CFG is replaced in the transformed CFG with an edge to a new, childless basic block that asserts the invariant for the associated loop. Otherwise, the CFGs are identical. This is illustrated in Figure 1(b) for the program of Figure 1(a), where invariant ϕ is left unspecified. Cutting every loop in a CFG leads to a loop-free CFG, for which verification conditions can be computed using weakest preconditions (an efficient method for this step is the main contribution of [1]). These verification conditions can then be discharged to a theorem prover, and if they are proven, the program is deemed correct. In Figure 1(b), taking ϕ to be $(a \neq b \wedge b \neq c \wedge c \neq a)$ allows a proof of correctness to succeed. The main problem with the inductive invariant approach is finding the required loop invariants.

[1] A variable is *havocked* if it is assigned a nondeterministic value. A *loop-modified variable* is a variable that is the target of an assignment in the loop under consideration.

Our contribution: combined-case k-induction. In *combined-case* k-induction, the strengths of k-induction and the inductive invariant approach are brought together. Like the inductive invariant approach, combined-case k-induction works by cutting loops in the input CFG one at a time, resulting in a single program that needs to be checked, but due to the use of a strong form of induction no external invariant is required.

A non-negative integer k_L is associated with each loop L in the input CFG. Loop L is then k_L*-cut* by replacing it with: a *base case*, which consists of k_L copies of the loop body and checks that no assertion can be violated within k_L loop iterations; and a *step case*, in which first all loop-modified variables are havocked, followed by k_L copies of the loop body where all assertions are replaced with assumptions (removing all edges exiting the loop), and finally a regular copy of the loop body, in which back edges to the loop header are removed. This is illustrated in Figure 1(c) (for $k_L = 3$) for the CFG of Figure 1(a): the left half of Figure 1(c) constitutes the base case, whereas the right blocks belong to the step case.

As in the inductive invariant approach, verifying the correctness of the loop-free program in Figure 1(c) implies that also the original program (Figure 1(a)) is correct. k-Induction, however, has the advantage that verification may succeed using weaker loop invariants. In fact, we can observe that the program in Figure 1(c) is correct; unlike the inductive invariant approach, no external invariant (like $a \neq b \wedge b \neq c \wedge c \neq a$) is required. Intuitively, the use of k-induction has implicitly strengthened the assertion $a \neq b$, resulting in an invariant sufficient to verify the program. We also say that the formula $a \neq b$, albeit not inductive, is *3-inductive*. Further, note that the final assertion $x = 0$ can be verified due to the fact that x is not assigned to in the loop; with earlier forms of k-induction that generate separate base and step cases, verifying this assertion would require auxiliary invariants to be provided.

References

[1] Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87. ACM, 2005.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In *CASSIS*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[3] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[4] Dino Distefano and Matthew J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226. ACM, 2008.

[5] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. In *Proceedings of the 18th International Static Analysis Symposium (SAS'11)*, volume 6887 of *LNCS*, pages 351–368. Springer, 2011.

[6] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of scratch-pad memory code for heterogeneous multicore processors. In *TACAS*, volume 6015 of *LNCS*, pages 280–295. Springer, 2010.

[7] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of DMA races using model checking and k-induction. *Formal Methods in System Design*, 2011. To appear, DOI: 10.1007/s10703-011-0124-2.

[8] R.W. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19—32. AMS, 1967.

[9] K. Rustan M. Leino. Dafny: an automatic program verifier for functional correctness. In *LPAR (Dakar)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[10] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

Verification and Code Generation for Invariant Diagrams in Isabelle

Viorel Preoteasa and Ralph-Johan Back and Johannes Eriksson

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3-5A, 20520 Åbo, Finland

Introduction. *Invariant-based programming (IBP)* is a correct-by-construction programming methodology in which the invariants of a program are developed before the code [1]. We describe here a method for fully automatic translation of an invariant diagram into a functional program, together with the associated consistency, termination and liveness proofs. The generated theorems are the obligations derived using the proof rules of invariant diagrams. The generated program is a refinement of the invariant diagram, and can be directly converted to executable code by Isabelle. This allows verified compilation of invariant diagrams into any of the languages supported by Isabelle code generator.

Invariant diagrams. Programs in IBP are expressed as *invariant diagram*s. An invariant diagram is a directed graph of *situations* (labeled predicates) connected by *transitions* (guarded assignment statements). Figure 1 shows an invariant diagram representing a program that searches for the first occurrence of the value x (of the generic type $'a$) in an array a (of type $nat \to 'a$). We use throughout mathematical notation close to the Isabelle syntax; e.g., application of function a to argument i is denoted by $a\ i$.

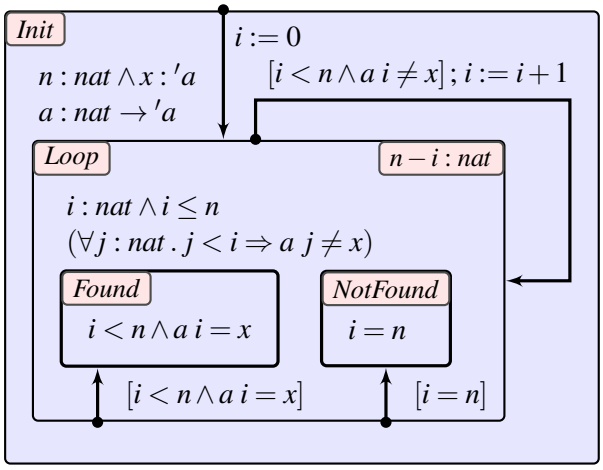


Figure 1: Searching for the first occurrence of x in a

The situations in Figure 1 are *Initial*, *Loop*, *NotFound*, and *Found*. Nesting means strengthening the predicate; e.g., the invariant of situation *Found* is the conjunction of all predicates in *Initial*, *Loop* and *Found*. A transition from situation s_1 to situation s_2, guarded by the predicate g, and assigning the value e to the variable x is written as $[g]; x := e$ on the arrow between s_1 and s_2. When omitted, the guard defaults to `true`. The execution starts from an initial situation and follows the transitions which are *enabled* (the guards are true). At each step the invariant of the current situation must be satisfied by the current value of the program variables. The execution terminates in the current situation if there are no enabled transitions. We note that an invariant diagram can have multiple postconditions: the above program terminates in *Found* if the value x is found in the array, and otherwise it terminates in *NotFound*. A diagram is *correct* iff each transition establishes its target situation (*consistency*), execution does not terminate in a situation other than the postconditions (*liveness*), and there are no infinite loops (*termination*). Sound and complete Hoare like proof rules for invariant diagrams are given in [2].

Isabelle/HOL. In the sequel, we describe an embedding of the program in Figure 1 into the logical framework of Isabelle/HOL. Isabelle is a theorem prover developed by Nipkow et al [6]. It includes a higher-order logic (HOL) deduction system, proof language (Isar), and a resolution-based prover. Notably, Isabelle supports modular locales for dealing with parametric theories and interpretations [3], mutually recursive function definitions, and a code generation facility with OCaml, ML, Scala and Haskell targets [5].

Invariant representation. An Isabelle locale declaration introduces a local scope of constants, assumptions, definitions, and theorems. We encode each one of the situations of the program in Figure 1 as a locale as follows:

```
locale Init =
  fixes n : nat
  and a : nat → 'a
  and x : 'a
```

```
locale Loop = Init +
  fixes i : nat
  assumes i ≤ n
  and (∀j. j < i ⇒ a j ≠ x)
```

```
locale Found = Loop +
  assumes i < n
  and a i = x
```

```
locale NotFound = Loop +
  assumes i = n
```

Program variables introduced in a situation translate into constants in the corresponding locale, whereas the predicates translate into assumptions. Nesting translates naturally to locale extension.

Proof obligations: consistency, termination and liveness An invariant diagram is consistent if for every transition $[g]; x := e$, from a situation with predicate P to a situation with predicate Q, the condition $P \wedge g \Rightarrow Q[x := e]$ is true, where $Q[x := e]$ is the substitution of the variable x in Q by expression e. We express these proof obligations in Isabelle by interpretation of the above locales. For instance, the transitions from *Loop* generate the following lemmas:

```
lemma tr_loop_loop:
  assumes start: Loop n a x i
  and grd: i < n ∧ a i ≠ x
  shows Loop n a x i + 1
```

```
lemma tr_loop_found:
  assumes start: Loop n a x i
  and grd: i < n ∧ a i = x
shows Found n a x i
```

```
lemma tr_loop_not_found:
  assumes start: Loop n a x i
  and grd: i = n
  shows NotFound n a x i
```

The loop transition additionally introduces an obligation that the termination function $n - i$, with lower bound 0, is decreased by the assignment $i := i + 1$:

```
lemma termination: i < n ∧ a i ≠ 0 ⇒ (n − i)[i := i + 1] < n − i
```

Finally, liveness entails that the disjunction of the guards from situations that are not postconditions is always true. In our example, the transition from *Init* is unconditional, and the disjunction $(i < n \wedge a\ i = x) \vee (i = n) \vee (i < n \wedge a\ i \neq x)$ follows from the invariant of *Loop*. The above obligations are expected to be proved by the programmer as part of the IBP workflow.

Program definition in Isabelle. The transition graph of the program in Figure 1 translates into the following collection of mutually recursive functions:

```
function
    init_fun n a x        =  loop_fun n a x 0
    loop_fun n a x i      =  if i < n ∧ a i ≠ x then loop_fun n a x (i+1)
                             else if i < n ∧ a i = x then found_fun n a x i
                             else if i = n then notfound_fun n a x i
                             else (i, Loop)
    found_fun n a x i     =  (i, Found)
    notfound_fun n a x i  =  (i, NotFound)
```

Each function corresponds to a situation in the invariant diagram; the parameters are the variables defined in the corresponding situation. Each function returns a pair of an integer i and a situation index s. In general the functions associated to a diagram return a tuple containing values for all variables updated by the diagram and additionally the situation index. The values returned by a function corresponding to a situation s are the values that would be computed by the diagram as described earlier when starting from s. The situation component of the result stores the final situation in which the termination occurred. For example, the execution of the search diagram starting from the init situation is equivalent to computing $(i, s) =$ `init_fun` $n\ a\ x$. The termination proof for this mutually recursive function definition can be constructed mechanically using the theorem `termination`.

We define the predicate for the total correctness of the diagram also as a locale.

```
locale Correctness =
  fixes n, a, x, i, s
  assumes s = Init ⇒ false
  and s = Loop ⇒ false
  and s = NotFound ⇒ NotFound n a x i
  and s = Found ⇒ Found n a x i
```

In this locale the constant s represents the situation in which the execution of the diagram terminates. The locale states the predicate that must be true at the end of the execution. If for example $s = NotFound$, then the final values must satisfy the locale `NotFound`. If $s = Init$ or $s = Loop$, then the final values must satisfy false, which is not possible. This means that the program is live, it would never terminate in these two situations. Next theorem states the correctness of the program.

```
theorem correctness:
  Init n a x ⇒ let (j, s) = init_fun n a x in Correctness n a x j s
  and Loop n a x i ⇒ let (j, s) = loop_fun n a x i in Correctness n a x j s
  and Found n a x i ⇒ let (j, s) = found_fun n a x i in Correctness n a x j s
  and NotFound n a x i ⇒ let (j,s) = notfound_fun n a x i in Correctness n a x j s
```

If for example n, a, and x satisfy the `Init` locale, then the final values calculated by the program when started in the initial situation satisfy the `Correctness` locale. The proof of this theorem can be mechanically constructed by applying the *induction theorem* generated by Isabelle from the definition of the mutually recursive functions, followed by applying the lemmas for the transitions, and for liveness.

Code generation. We can invoke the built-in code generator in Isabelle to export the functions defined above to any of the supported target languages. For instance, the Haskell rendition of `loop_fun` is:

```
loop_fun n a x i =
  (if less_nat i n && not (a i == x) then loop_fun n a x (plus_nat i one_nat)
  else (if less_nat i n && a i == x then found_fun n a x i
  else (if equal_nat i n then notfound_fun n a x i
  else (i, Loop))));
```

As Haskell optimizes tail recursion, the program remains iterative throughout translation.

Conclusion and future work. We have described a translation validation-based approach to both verification and code generation for IBP. A program is (mechanically) translated into an Isabelle/HOL theory containing the invariants (as locales), the proof obligations of the program, a functional representation, and consistency and liveness theorems stating that the functional representation terminates in the intended postcondition. The proof obligations should be proved by the programmer; they become lemmas for the (automatically constructed) consistency and liveness proofs for the functional representation. The functional representation can be directly translated into executable code by Isabelle. As future work, we plan to automate the translation in Socos [4], a verification tool for invariant diagrams. The current translation does not assume the loop invariant in the termination proof. For more realistic examples, this will be required together with more general terminating functions to handle, e.g., nested loops.

References

[1] Ralph-Johan Back. Invariant based programming: Basic approach and teaching experiences. *Formal Aspects of Computing*, 21(3):227–244, 2009.

[2] Ralph-Johan Back and Viorel Preoteasa. Semantics and proof rules of invariant based programs. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 2011.

[3] Clemens Ballarin. Locales and locale expressions in isabelle/isar. In *TYPES FOR PROOFS AND PROGRAMS (TYPES 2003), LNCS 3085*, pages 34–50. Springer, 2004.

[4] Johannes Eriksson and Ralph-Johan Back. Correct-by-construction programming in the Socos (2) environment. In *THedu'11 (CTP Components for Educational Software), workshop associated to CADE'2011*.

[5] Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming (FLOPS 2010)*, volume 6009, 2010.

[6] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

A Proof System for Adaptable Class Hierarchies

Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu
Department of Informatics, University of Oslo, Norway
{johand,einarj,olaf,ingridcy}@ifi.uio.no

An intrinsic property of software in the real world is that it needs to evolve. This can be as part of the initial *development* phase, *improvements* to meet new requirements, or as part of a software *customization* process such as, e.g., feature selection in software product lines or delta-oriented programming [1]. As the code is enhanced and modified, it becomes more complex and drifts away from its original design [9]. For this reason, it may be desirable to redesign the code base to improve its structure, thereby reducing software complexity. For example, the process of *refactoring* in object-oriented software development describes changes to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior [5]. In this paper, the term *adaptable class hierarchies* covers transformations of classes during the development, improvement, customization, and refactoring of class hierarchies.

Reasoning about properties of object-oriented systems is in general non-trivial due to complications including class inheritance and late binding of method calls. Object-oriented software development is based on an open world assumption; i.e., class hierarchies are typically extendable. In order to have reasoning control under such an open world assumption, it is advantageous to have a framework which controls the properties required of method redefinitions. With a *modular reasoning* framework, a new subclass can be analysed in the context of its superclasses, such that the properties of the superclasses are guaranteed to be maintained. This has the advantage that each class can be fully verified at once, independent of subclasses which may be designed later. The best known modular framework for class hierarchies is *behavioral subtyping* [8]. However, behavioral subtyping has been criticized for being overly restrictive and is often violated in practice [10].

Incremental reasoning frameworks generalize modular reasoning by possibly generating new verification conditions for superclasses in order to guarantee established properties. Additional properties may be established in the superclasses after the initial analysis, but old properties remain valid. Observe that these frameworks subsume modularity: if the initial properties of the classes are sufficiently strong (for example by adhering to a behavioral contract), it will never be necessary to add new properties later. *Lazy behavioral subtyping* is a formal framework for such incremental reasoning, which allows more flexible code reuse than modular frameworks. The basic idea underlying lazy behavioral subtyping is a separation of concerns between the behavioral *specifications* of method definitions from the behavioral *requirements* to method calls. Both specifications and requirements are manipulated through a bookkeeping framework which controls analysis and proof obligations in the context of a given class. Properties are only inherited by need. Inherited requirements on method redefinition are as weak as possible for ensuring soundness. Lazy behavioral subtyping seems well-suited for the incremental reasoning style desirable for object-oriented software development, and can be adjusted to different mechanisms for code reuse. It was originally developed for single inheritance class hierarchies [3], but has later been extended to multiple inheritance [4] and to trait-based code reuse [2].

Adaptable class hierarchies add a level of complexity to proof systems for object-oriented programs, as superclasses in the middle of a class hierarchy can change. Unrestricted, such changes may easily violate previously verified properties in both sub- and superclasses. The management of verification conditions becomes more complicated than in the case of extending a class hierarchy at the bottom with new subclasses. We consider a version of Featherweight Java [6] extended with behavioral interfaces, in which methods are annotated with pre/postconditions, and consider a number of *basic update operations* for adapting class definitions. We consider a series of "snapshots" of a class hierarchy during a develop-

ment and adaptation process, in which the developer applies class updates and analysis steps to the class hierarchy.

The paper presents a calculus which tracks verification conditions when the program changes, based on an incremental reasoning framework. In our approach, a requirement holds if it has a proof (or follows from another proven property). The basic update operations defined in this paper include addition, removal, and redefinition of code, and may be composed to form more complex modifications, e.g., the Pull Up Method refactoring [5] is illustrated by an example. The work in this paper extends [7], which deals with static typing, to handle behavioral specifications of methods. To allow incremental reasoning in a flexible manner, we extend the lazy behavioral subtyping mechanism to deal with program transformations not limited to behavioral preserving evolution. Consider some method *m*, originally equipped with a pre/post specification. As the system evolves, methods that call *m* may be modified such that *m* is called with weaker requirements, and calls to *m* may even disappear. If *m* is later changed, the new version need not adhere to the original specification of *m*; the new version only needs to respect the behavior required by call-sites *at the time* when the modification of *m* is performed. The specified and required properties of method definitions and method calls are tracked through the adaptation steps. We prove soundness of the proposed verification system.

A complementary line of work is *proof adaptation*, which has been studied in the automated theorem prover communities. Proof adaptation considers how to adapt a proof when a specification changes. Such techniques would fit naturally into an implementation of the reasoning framework proposed in this paper, to alleviate the overhead of additional proof activity resulting from the flexibility of the proposed framework.

References

[1] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In M. Bernardo and V. Issarny, editors, *Proc. 11th Intl. School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM 2011)*, LNCS 6659, pages 417–457. Springer, 2011.

[2] F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer. Verifying traits: A proof system for fine-grained reuse. In *Proc 13th Workshop on Formal Techniques for Java-like Programs (FTfJP'11)*. ACM, 2011. To appear.

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

[4] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Incremental reasoning with lazy behavioral subtyping for multiple inheritance. *Science of Computer Programming*, 76(10):915–941, 2011.

[5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Aug. 1999.

[6] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[7] E. B. Johnsen, M. Kyas, and I. C. Yu. Dynamic classes: Modular asynchronous evolution of distributed concurrent objects. In A. Cavalcanti and D. Dams, editors, *Proc. 16th International Symposium on Formal Methods (FM'09)*, LNCS 5850, pages 596–611. Springer, 2009.

[8] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, Nov. 1994.

[9] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[10] N. Soundarajan and S. Fridella. Inheritance: From code reuse to reasoning reuse. In P. Devanbu and J. Poulin, editors, *Proc. Fifth International Conference on Software Reuse (ICSR5)*, pages 206–215. IEEE Computer Society Press, 1998.

Refinement for Open Mixed Trees

Marco Carbone
IT University of Copenhagen
Copenhagen, Denmark
carbonem@itu.dk

Thomas Hildebrandt
IT University of Copenhagen
Copenhagen, Denmark
hilde@itu.dk

Hugo A. López
IT University of Copenhagen
Copenhagen, Denmark
lopez@itu.dk

Abstract

We propose a new denotational behavioral model called *open mixed trees* which generalizes standard model of labelled trees (where labels are marked as negative, positive or both) by annotating each state with a set of so-called *open* actions and a flag indicating if termination is allowed in the state or not. We then propose a generalization of covariant-contravariant simulation that also takes account of termination and allows intermediate open parts of the specification.

1 Introduction

Consider a healthcare workflow process in which you have a patient Alice, a doctor Bob, and a Social Worker Charlie. The following set of activities are included in the first specification S:

1. Alice comes to Bob for a medical appointment.
2. Bob receives Alice and gathers her symptomatology.
3. After consultation, Bob formulates a medicine treatment for Alice.
4. Bob sends the medicine formulation to Charlie, so he can deliver it to Alice.
5. Alice gets the medicine from Charlie and starts taking her treatment regularly as specified by Bob.
6. After some days, Alice comes back to Bob for a control, and the symptoms have disapeared.

Many details have been hindered from this example. First of all, it only details the interaction between three of the main actors involved. We may have a private health care institution that has to fulfill the auditing processes, where between activity 2 and 3. other actors will come into play. Our specification S could be extended accordingly to a new model S' including the two actions

- On insuficiency of information, Bob will take blood samples and supplementary tests from Alice.
- On cases with high variability, Bob will consult a pool of specialists on Alice's case.

between action 2. and 3.

It is to note, that even when S' has more behavior than S, it is still constrained to a set of activities that can be performed. The extra set of activities can be repeated many times and with different execution orders, but activities outside this set have to be ruled out. For instance, Bob cannot start operating Alice just after having gathered her symptomatology.

How is S related to S'? It is clear that the notion that we are looking for has a lot to do with the notion of *refinement*. In many cases we will specify systems by adding up more and more roles (and their respective behaviors) over the time. This, will lead us to start with a specification like S, knowing that each of the actions can be further refined with more and more behavior. We propose a new controlled way, called *open refinement*, to specify where and which actions can be inserted. The idea is in addition to standard transitions $P \xrightarrow{a} P'$ where a process P exhibits an immediate action a before evolving into P'

to also specify *open* states ${}_A\!\circlearrowright\! P \xrightarrow{a} P'$, where the process P can exhibit a finite series of actions in A before evolving with a into P'. The open state allows us to describe explicit stages in a process in which a process can be refined with any of the actions in a constrained set A. Here, transitions become weaker, as they might need more than one step for moving from P to P', but they also become broader than the standard weak transitions, as the set A can involve several (and possibly visible) actions and not just a dedicated internal action.

These changes lead us to proposing a new notion of refinement we call *open mixed refinement*. Starting from the covariant-contravariant simulations (that allow mixed, externally and internally controlled, activities and captures the necessary difference between such) we add the new notion of open states and also the ability to specify explicitly if a system may terminate in a state from which additional internally controlled activities are possible.

We believe the proposed model has both good uses in practice and good properties, i.e. can be given a clean categorical representation. We start in this brief abstract by giving the definition and the first result that open mixed refinement specializes to covariant-contravariant simulation if one allows no open states and always allows termination.

2 Open Mixed Trees and Refinement

Definition 1 (Open Mixed Trees)**.** *An* open mixed tree *is a tuple* $T = \langle S, s_0, \mathsf{Act}^-, \mathsf{Act}^+, \sigma, \rightarrow \rangle$ *where*

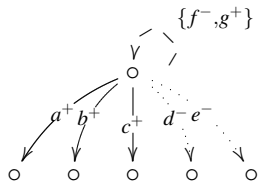
- S *is a set of* states*,*
- $s_0 \in S$ *is the* initial state*,*
- $\mathsf{Act} = \mathsf{Act}^- \cup \mathsf{Act}^+$ *is a set of actions characterized as externally controlled actions in* Act^- *(denoted by* a^-*) and internally controlled actions in* Act^+ *(denoted by* a^+*).*
- $\rightarrow \subseteq S \times \mathsf{Act} \times S$ *is a labelled transition relation between states*
- $\sigma : S \rightarrow \mathscr{P}(\mathsf{Act} \cup \{\checkmark\})$ *defines for each state the* open actions *and the possibility of terminating*
- $\checkmark \notin \sigma(s) \implies \exists s \xrightarrow{b^+}$*, i.e. an internally controlled action must be possible from every non-terminating state*
- $\forall s \in S$*, there exists a unique path* $S_0 \rightarrow^* s$ *(i.e. the transition relation forms a tree)*

An open mixed tree where $\sigma(s) = \{\checkmark\}$ *for all* $s \in S$*, i.e. an open mixed tree with no open actions and which allow termination in every state, is referred to as just a mixed tree. A mixed tree is equivalent to a normal tree labelled with positive and negative labels.*

Intuitively, an open mixed tree represents the specification of a reactive, non-deterministic system with both internally controlled actions (e.g. output) and externally controlled actions (e.g. input). Note that there may be actions in $\mathsf{Act}^- \cap \mathsf{Act}^+$ that are both externally and internally controlled.

In any state with at least one internally controlled action any implementation must be able to do at least one of the internally controlled actions, or terminate if termination is also allowed by the specification. The states in which it is allowed to terminate is defined by the set T. Note that in order to not have any contradictions a state which is not in T (i.e. termination without further internally controlled actions is not allowed) must have at least one internally controlled action out of it.

Finally, the function σ pairs each state with a set of *open* (or underspecified) behavior, which allows an implementation to perform any action within the set a finite number of times before progressing (or terminating if the state is in T. We can depict open trees easily:



Below we write $s_1 \xrightarrow{\ell^d} s_2$ when $\{s_1, \ell, s_2\} \in \rightarrow$ and $\ell \in \mathsf{Act}^d$. Similarly, we write $\sigma^+(s_1)$ for the set of transitions such that $s_1 \xrightarrow{a^+} s_1' \in \sigma$

Definition 2 (Refinement)**.** *A binary relation $\mathscr{R} \subseteq S_1 \times S_2$ between the state sets of two open mixed trees $P_j = \langle S_j, i_j, \mathsf{Act}^-, \mathsf{Act}^+, \sigma_j, \rightarrow_j \rangle$ for $j \in \{1,2\}$ is a refinement if $i_1 \mathscr{R} i_2$ and $s_1 \mathscr{R} s_2$ implies*

1. *$\forall s_1 \xrightarrow{a^-}_1 s_1'$, implies $\exists s_2 \xrightarrow{a_1}_2 s_{2,1} \xrightarrow{a_2}_2 \cdots \xrightarrow{a_n}_2 s_{2,n} \xrightarrow{a^-}_2 s_{2,n+1}'$, $a_i \in \sigma_1(s_1)$, and $s_1 \mathscr{R} s_{2,i}$*
2. *$\forall s_2 \xrightarrow{a^+}_2 s_2'$ implies (i) $\exists s_1 \xrightarrow{a^+}_1 s_1'$, and $s_1' \mathscr{R} s_2'$ or (ii) $a \in \sigma_1^+(s_1)$ and $s_1 \mathscr{R} s_2'$*
3. *$\sigma_2(s_2) \subseteq \sigma_1(s_1)$,*
4. *$\checkmark \in \sigma(s_1) \implies s_2 \xrightarrow{a_1}_2 s_{2,1} \xrightarrow{a_2}_2 \cdots \xrightarrow{a_n}_2 s_{2,n}$ and $a_i \in \sigma_1(s_1)$, $s_1 \mathscr{R} s_{2,i}$ and $\checkmark \in \sigma_s(s_{2,n})$.*
5. *if $s_2 = s_{2,0} \xrightarrow{a_0}_2 s_{2,1} \xrightarrow{a_1}_2 s_{2,2} \xrightarrow{a_2} \cdots$, $s_1 = s_{1,0}$ and ($s_{1,i} \xrightarrow{a_i} s_{1,i+1}$ or ($s_{1,i} = s_{1,1+1}$ and $a_i \in \sigma_1(s_{1,i})$) and $s_{1,i} \mathscr{R} s_{2,i}$ for $i \in \omega$, then $|\{s_{1,i}\}_{i \in \omega}| = \omega$.*

We say that Q is a refinement *of P, written $P \sqsubseteq Q$, whenever there exists a relation $\mathscr{R}$ such that $P \mathscr{R} Q$.*

Proposition 1. *The refinement relation $\sqsubseteq$ between open mixed trees as defined above*

1. *is reflexive and transitive, and*
2. *contains the identity relation*

As stated in the proposition below, refinement specializes for mixed trees (i.e. open mixed trees with no open actions and which allow termination in every state) to the notion of covariant-contravariant simulation defined in [2, 1].

Proposition 2. *Refinement for mixed trees coincides with $(\mathsf{Act}^+ \backslash \mathsf{Act}^-, \mathsf{Act}^- \backslash \mathsf{Act}^+)$-simulation as defined in [2], taking $\mathsf{Act}^- \cap \mathsf{Act}^-$ as the set of actions with "bi"-polarity, i.e. both internally and externally controlled.*

References

[1] L. Aceto, I. Fábregas, D. de Frutos Escrig, A. Ingólfsdóttir, and M. Palomino. Relating modal refinements, covariant-contravariant simulations and partial bisimulations. *Fundamentals of Software Engineering, FSEN*, 2011.

[2] I. Fabregas, D. de Frutos Escrig, and M. Palomino. Logics for contravariant simulations. In *Formal Techniques for Distributed Systems: Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010, Proceedings*, volume 6117, page 224. Springer-Verlag New York Inc, 2010.

Evaluation à la Carte

Non-Strict Evaluation via Compositional Data Types

Patrick Bahr
Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Abstract

We describe how to perform monadic computations over recursive data structures with fine grained control over the evaluation strategy. This solves the issue that the definition of a recursive monadic function already determines the evaluation strategy due to the necessary sequencing of the monadic operations. We show that compositional data types already provide the structure needed in order to delay monadic computations at any point of the computation.

1 Introduction

Algebraic data types offer an excellent representation of abstract syntax trees (ASTs). The ease with which functional programming languages allow us to manipulate algebraic data types makes the functional programming paradigm a powerful tool for performing transformations on ASTs – an ubiquitous tasks when writing compilers and interpreters.

As an example, consider the following *Haskell* [4] definition of an algebraic data type representing a simple expression language over integers and pairs:

$$\begin{aligned}&\textbf{data}\ \mathit{Exp} = \mathit{Const}\ \mathit{Int} \mid \mathit{Pair}\ \mathit{Exp}\ \mathit{Exp}\\ &\qquad\qquad \mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp} \mid \mathit{Fst}\ \mathit{Exp} \mid \mathit{Snd}\ \mathit{Exp}\end{aligned}$$

Apart from the constructors for integers and pairs, the language contains addition and the projection operators *Fst* and *Snd*. Implementing an evaluation function for this language is a simple exercise:

$$\begin{aligned}&\mathit{eval} :: \mathit{Exp} \to \mathit{Exp}\\ &\mathit{eval}\ (\mathit{Const}\ i) = \mathit{Const}\ i\\ &\mathit{eval}\ (\mathit{Pair}\ x\ y) = \mathit{Pair}\ (\mathit{eval}\ x)\ (\mathit{eval}\ y)\\ &\mathit{eval}\ (\mathit{Add}\ x\ y) = \textbf{case}\ (\mathit{eval}\ x, \mathit{eval}\ y)\ \textbf{of}\\ &\quad (\mathit{Const}\ i, \mathit{Const}\ j) \to \mathit{Const}\ (i+j)\\ &\mathit{eval}\ (\mathit{Fst}\ p) = \textbf{case}\ \mathit{eval}\ p\ \textbf{of}\ \mathit{Pair}\ x\ y \to x\\ &\mathit{eval}\ (\mathit{Snd}\ p) = \textbf{case}\ \mathit{eval}\ p\ \textbf{of}\ \mathit{Pair}\ x\ y \to y\end{aligned}$$

While this function performs the desired evaluation, its type is not as precise as we would expect. According to its type, *eval* produces an expression of type *Exp* which potentially can contain additions and projections. This can be solved by using as codomain of *eval* a separate type *Value* that only contains (copies of) the constructors *Const* and *Pair*. This means, however, that also code that works on both *Exp* and *Value* has to be duplicated, e.g. pretty printing and parsing.

2 Data Types à la Carte

Swierstra's *data types à la carte* [5] offer an elegant solution to this problem by representing expression types as a fixed point of a functor:

$$\textbf{data}\ \mathit{Term}\ f = \mathit{Term}\ f\ (\mathit{Term}\ f)$$

This approach makes it possible to define the signature of the expression language in two components – values and operations – and combine them via the formal sum $\oplus$ of functors:

$$\textbf{data}\ (f \oplus g)\ e = \mathit{Inl}\ (f\ e) \mid \mathit{Inr}\ (g\ e)$$

We can then define the signatures of our expression language as follows:

$$\begin{aligned}&\textbf{data}\ \mathit{Value}\ e = \mathit{Const}\ \mathit{Int} \mid \mathit{Pair}\ e\ e\\ &\textbf{data}\ \mathit{Op}\ e = \mathit{Add}\ e\ e \mid \mathit{Fst}\ e \mid \mathit{Snd}\ e\\ &\textbf{type}\ \mathit{Sig} = \mathit{Op} \oplus \mathit{Value}\end{aligned}$$

This allows us to represent values and expressions as *Term Value* and *Term Sig*, respectively.

In addition, Swierstra also defines a binary *type class* $\prec$ on signature functors that approximates inclusion. That is, $f \prec g$ if g is equal to f or contains

it as a summand. Most importantly, this type class provide a function to inject a "smaller" signature into a "bigger" one:

$$inject :: (f \prec g) \Rightarrow f\ (Term\ g) \to Term\ g$$

Functions of the form $Term\ f \to r$ are written as *catamorphisms* induced by algebras, i.e. functions of type $f\ r \to r$. This allows us to write functions on a per signature basis, which is achieved by using a type class:

```
class Eval f where
  evalAlg :: f (Term Value) → Term Value
```

The instantiation of this class for values is trivial:

```
instance Eval Value where
  evalAlg = inject
```

For operator symbols we have to provide an implementation that evaluates the arguments appropriately:

```
instance Eval Op where
  evalAlg (Add x y) = case (x, y) of
    (Term (Const i), Term (Const j))
      → inject (Const (i + j))
  evalAlg (Fst p)   = case p of
    Term (Pair x y) → x
  evalAlg (Snd p)   = case p of
    Term (Pair x y) → y
```

Note that the case distinction in the above evaluation algebra as well as in the direct evaluation in Section 1 is incomplete: In case that an argument is not of the expected type, e.g. *Fst* is applied to an integer constant, the evaluation halts with a runtime error.

3 Monadic Algebras and Thunks

In order to recover from runtime errors, it is better to use monads to indicate failure explicitly. This can be easily achieved by defining a monadic algebra [3, 1], i.e. a function of type $f\ r \to m\ r$ for a monad m. Such a monadic algebra gives rise to a monadic catamorphism of type $Term\ f \to m\ r$. The evaluation algebra from Section 2 can be easily adapted to such a monadic style. Unfortunately, this will determine the evaluation strategy: The arguments of the operator symbols such as *Fst* have to be evaluated to *normal form* (in order to determine whether an error occurred). For example, the evaluation of an expression of the form $fst\ (3, snd\ 5)$ will yield an error since the evaluation of $(3, snd\ 5)$ to normal form fails due to the second component of the pair.

In order to regain control over the evaluation strategy, we have to allow arbitrary nesting of monadic computations in the result. Instead of the monadic result type $m\ (Term\ Value)$, we therefore use the result type $Term\ (m \oplus Value)$ – making the monad part of the target signature. A monadic computation can thus be embedded into the term structure.

```
thunk :: m (Term (m ⊕ f)) → Term (m ⊕ f)
thunk = inject
```

The evaluation of terms with such *thunks* to *weak head normal form* (*whnf*) is implemented by sequencing all thunks until a proper constructor (i.e. in the f-part of the signature) is reached:

```
whnf :: Monad m ⇒
        Term (m ⊕ f) → m (f (Term (m ⊕ f)))
whnf (Term (Inl m)) = m >>= whnf
whnf (Term (Inr t)) = return t
```

We can now use this idea to define a non-strict monadic evaluation function using the *Maybe* monad to indicate failure:

```
class EvalT f where
  evalAlgT :: f (Term (Maybe ⊕ Value))
           → Term (Maybe ⊕ Value)
```

Again, the case for the value constructors is trivial:

```
instance EvalT Value where
  evalAlgT = inject
```

For evaluating the operator symbol applications, we simply evaluate their arguments to whnf and create a thunk in the end:

```
evalAlgT (Add x y) = thunk $ do
  Const i ← whnf x
  Const j ← whnf y
```

```
    return (inject (Const (i+j)))
evalAlgT (Fst v)    = thunk $ do
    Pair x y ← whnf v
    return x
evalAlgT (Snd v)    = thunk $ do
    Pair x y ← whnf v
    return y
```

By constructing the catamorphism of this algebra, we obtain the following evaluation function

```
evalT :: Term Sig → Term (Maybe ⊕ Value)
evalT = cata evalAlgT
```

With only mild assumptions on the signature functors, we can also easily implement the evaluation to normal form by simply iterating the *whnf* function:

```
nf :: (Monad m, Traversable f) ⇒
      Term (m ⊕ f) → m (Term f)
nf = liftM Term . mapM nf <=< whnf
```

Eventually, we obtain the desired non-strict evaluation function:

```
eval :: Term Sig → Maybe (Term Value)
eval = nf . evalT
```

Using this evaluation function, the expression *fst* $(3, snd\ 5)$ now evaluates to the expected value 3.

Full non-strict evaluation, however, is only one option that we now have. We can stipulate additional strictness if desired, similarly to Haskell's strictness annotations. The following function makes every constructor strict by evealuating each of its arguments to whnf:

```
strict :: (f ≺ g, Traversable f, Monad m) ⇒
          f (Term (m ⊕ g)) → Term (m ⊕ g)
strict = thunk . liftM inject .
           mapM (liftM inject . whnf)
```

Now we can, for example, make all value constructors strict simply by replacing *inject* with *strict*:

```
instance EvalT Value where
  evalAlgT = strict
```

We can even be more specific: It is possible to define the following combinator, which takes a specification of which arguments are supposed to be strict and then performs the desired evaluation strategy:

```
strictAt :: (f ≺ g, Traversable f, Monad m, ...) ⇒
            (∀ a . Ord a ⇒ f a → [a]) →
            f (Term (m ⊕ g)) → Term (m ⊕ g)
```

For example, we can make only the second component of the *Pair* value constructor strict:

```
instance EvalT Value where
  evalAlgT = strictAt spec
    where spec (Pair a b) = [b]
          spec _          = []
```

In a similar manner also other combinators can be formed that allow to specify the evaluation strategy in a very fine grained fashion.

4 Conclusions

This simple observation shows yet another useful aspect of using compositional data types as a framework for dealing with abstract syntax trees [1, 2].

In addition to the example presented here, we have applied similar techniques to also control the evaluation strategy for other recursion schemes such as tree homomorphisms, tree transducers and attribute grammars.

References

[1] Patrick Bahr and Tom Hvitved. Compositional data types. WGP 2011, to appear.

[2] Laurence Day and Graham Hutton. Towards Modular Compilers For Effects. In *Proceedings of the Symposium on Trends in Functional Programming*, Madrid, Spain, 2011.

[3] Maarten Fokkinga. Monadic Maps and Folds for Arbitrary Datatypes. Technical report, Memoranda Informatica, University of Twente, 1994.

[4] Simon Marlow. Haskell 2010 Language Report, 2010.

[5] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.

Does it pay to extend the parameter of the world model?

Werner Damm

University of Oldenburg and OFFIS, Germany

(joint work with Bernd Finkbeiner, University Saarbrücken)

Abstract

Will the cost for observing additional real-world phenomena in a world model be recovered by the resulting increase in the quality of the implementations based on the model? We address the quest for optimal models in light of industrial practices in systems engineering, where the development of control strategies is based on combined models of a system and its environment. We introduce the notion of remorsefree dominance between strategies, where one strategy is preferred over another if it outperforms the other strategy in comparable situations, even if neither strategy is guaranteed to achieve all objectives. We call a world model optimal if it is sufficiently precise to allow for a remorsefree dominating strategy that is guaranteed to remain dominant even if the world model is refined. We present algorithms for the automatic verification and synthesis of dominant strategies, based on tree automata constructions from reactive synthesis.

Towards a Behavioral Analysis of Computer Algebra Programs*

(Extended Abstract)

Muhammad Taimoor Khan
DK Computational Mathematics
Johannes Kepler University
Linz, Austria
muhammad.khan@dk-compmath.jku.at

Wolfgang Schreiner
Research Institute for Symbolic Computation
Johannes Kepler University
Linz, Austria
Wolfgang.Schreiner@risc.jku.at

We present our initial results on the behavioral analysis of computer algebra programs. Computer algebra programs written in symbolic computation languages such as Maple and Mathematica sometimes do not behave as expected [5], e.g. by triggering runtime errors or delivering wrong results. There has been a lot of research on applying formal techniques to classical programming languages, e.g. Java [7], C# [1] and C [3] etc., but we aim to apply the same techniques to computer algebra languages. Therefore our goal is to design and develop a tool for the static analysis of computer algebra programs [12]. The tool will automatically find errors in programs annotated with extra information such as variable types and method contracts [10], in particular type inconsistencies and violations of method preconditions.

The task of applying formal techniques to widely used computer algebra languages (Maple and Mathematica) is more complex as these are fundamentally different from classical languages. In particular, we found the following challenges respectively differences to classical languages for formal type checking respectively specifying Maple programs (which are typical for most computer algebra languages):

- The language supports some non-standard types of objects, e.g. symbols, unevaluated expressions and polynomials.
- There is no clear difference between declaration and assignment. A global variable is introduced by an assignment; a subsequent assignment may modify the type information for the variable.
- The language uses type information to direct the flow of control in the program, i.e. it allows some runtime type-tests which selects the respective code-block for further execution. This makes type inference more complex.
- The language allows runtime type checking by type annotations but these annotations are optional which give rise to type ambiguities. This also makes type inference more complex.
- Maple values are organized in a kind of polymorphic type system with a sub-typing relationship such that we can assign a value to different types. This also makes type inference more complex.

The challenge for a specification language for Maple is to overcome those particularities of the language that hinder static analysis because Maple was not designed for this purpose (type checking respectively verification).

There are various computer algebra languages, Mathematica and Maple being the most widely used by far [13], both of which are dynamically typed. We have in our work chosen Maple for the following reasons:

- Maple has an imperative style of programming while Mathematica has a rule-based programming style with more complex semantics.

*The research was funded by the Austrian Science Fund (FWF): W1214-N15, project DK10. .

- Maple has type annotations for runtime checking which can be directly applied for static analysis. (There are also parameter annotations in Mathematica but they are used for selecting the appropriate rule at runtime).

Still the results we derive with type checking Maple can be applied to Mathematica, as Mathematica has almost the same kinds of runtime objects as Maple.

As a starting point, we have defined a subset of the computer algebra language Maple called *MiniMaple* [8, 9]. Since type safety is a prerequisite of program correctness, we have formalized a type system for *MiniMaple* and implemented a corresponding type checker. Furthermore, we have defined a specification language to formally specify the behavior of *MiniMaple* procedures and implemented a corresponding type checker. As the next step, we will develop a tool to automatically detect errors in *MiniMaple* programs with respect to their specifications.

In the following we will brief the main features of our work.

A Type System for *MiniMaple*: *MiniMaple* uses Maple type annotations for static analysis. Based on these annotations we defined a language of types and a corresponding type system. The type system supports the usual concrete data types, sets, lists and records. It also supports some non-standard types, e.g. the union type of various types, symbols, unevaluated expressions and polynomials etc. Type *anything* is the super-type of all types. The problem of statically type-checking *MiniMaple* programs is related to the problem of statically type-checking scripting languages such as Ruby [11], but there are also fundamental differences due to the different language paradigms.

In the following, we highlight the problems arising from type checking various *MiniMaple* programs.

- Global variables (declarations) can not be type annotated; therefore to global variables values of arbitrary types can be assigned in Maple. We introduce *global* and *local* contexts to handle the different semantics of the variables inside and outside of the body of a procedure respective loop.
  - In a *global* context new variables may be introduced by assignments and the types of variables may change arbitrarily by assignments.
  - In a *local* context variables can only be introduced by declarations. The types of variables can only be *specialized* i.e. the new value of a variable should be a sub-type of the declared variable type.
  - The sub-typing relation is observed while specializing the type of a variable.
- Maple supports type tests (i.e. type(I,T)) to direct the control flow of a program. Different branches of a conditional may have different pieces of type information for the same variable. We keep track of the type information introduced by the branches to allow only satisfiable tests.
- With the use of type-tests, the number of loop iterations might influence the type information and one cannot determine the concrete type by the static analysis. To handle this non-determination of types we put a reasonable upper bound (least fixed point) on the types of variables. As a special case this upper bound is the type of a variable prior to the body of a loop.

The type checker has been applied to the Maple package *DifferenceDifferential* [4]. No crucial typing errors have been found but some bad code parts have been identified that can cause problems.

A Specification Language for *MiniMaple*: Based on the formalism of our type system we have defined a formal specification language for *MiniMaple*. The specification language is a logical formula language that mainly uses Maple notations but also has its own notations. The language allows to formally specify the behavior of the procedures as a state relationship, e.g. by specifying pre/post-conditions of a procedure and other constraints. The specification language supports specification declarations, procedure and loop specifications and assertions. The language also supports abstract data types, while

the existing specification languages are weaker in such specifications. Currently we are defining formal semantics of *MiniMaple*. Based on this semantics we will define formal semantics of the specification language so that it specifies the intended algebraic properties. The specification language aims to realize respectively bridge the gap between actual computer algebra algorithm and its corresponding implementation [4].

We may use this specification language to generate executable assertions that are embedded in *MiniMaple* programs and check at runtime the validity of pre/post conditions. Our main goal, however, is to use the specification language for static analysis, in particular to detect violations of method preconditions. Here we currently investigate two possibilities:

1. We may directly generate verification conditions and use Satisfiability Modulo Theories (SMT) solvers or interactive theorem provers to prove their correctness.

2. We may use some existing framework to generate verification conditions and prove the correctness, e.g. by the Boogie [2] framework developed by Microsoft and Why [6] by LRI-France. Here we need to translate our specification annotated *MiniMaple* program into an intermediate language of Boogie/Why and then use their various proving back-ends for verification.

The formal specification of the *DifferenceDifferential* package developed at our institute will be the main test for our specification language and checking framework.

References

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS '04), LNCS*, volume 3362, pages 49–69. Springer, 2004.

[2] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, LNCS*, volume 4111, pages 364–387. Springer, 2006.

[3] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008. `http://frama-c.com/download/acsl_1.2.pdf`.

[4] Christian Dönch. Bivariate Difference-Differential Dimension Polynomials and Their Computation in Maple. Technical Report, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2009.

[5] Richard J. Fateman. Why Computer Algebra Systems Sometimes Can't Solve Simple Equations. *SIGSAM Bulletin*, 30(2):8–11, 1996.

[6] Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification condition generator. Research report 1366, LRI - CNRS UMR 8623, Université Paris-Sud, France, March 2003.

[7] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. A Tutorial, 2006. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`.

[8] Muhammad Taimoor Khan. Software for *MiniMaple*. `http://www.risc.jku.at/people/mtkhan/dk10/`.

[9] Muhammad Taimoor Khan. A Type Checker for *MiniMaple*. RISC Technical Report 11-05, also DK Technical Report 2011-05, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2011.

[10] Bertrand Meyer. Applying Design by Contract. *Computer*, 25:40–51, October 1992.

[11] Jeffrey S. Foster Michael Furr, Jong-hoon (David) An and Michael Hicks. Static Type Inference for Ruby. In *Proceedings of the 24th Annual ACM Symposium on Applied Computing, OOPS track*, Honolulu, HI, 2009.

[12] Wolfgang Schreiner. Project Proposal: Formally Specified Computer Algebra Software. Doktoratskolleg (DK), Johannes Kepler University, Linz, Austria, `http://www.risc.jku.at/projects/dk10`, 2007.

[13] Inna K. Shingareva and Carlos Lizárraga-Celaya. *Maple and Mathematica: A Problem Solving Approach for Mathematics*. Springer, 2nd ed. edition, September 2009.

Integrating Resource-Restricted Execution Contexts with Abstract Behavioral Specifications

Einar Broch Johnsen, Rudolf Schlatte, and S. Lizeth Tapia Tarifa
Department of Informatics, University of Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

1 Introduction

Formal approaches to performance analysis has traditionally been in the domain of embedded systems (for an overview, see [6]). However, the virtualization of resources in, e.g., cloud computing makes performance analysis at the modeling stage important also for general software in order to gain early insights into the resource needs of a service or component. Autonomous and self-managing software may even dynamically adapt its resource needs in response to changes in client behavior.

In order to specify, analyze, and predict non-functional system behavior at an early stage in the software development process, models need to naturally capture and range over relevant deployment scenarios. For this reason, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language and integrate different execution contexts with the behavioral model. We propose an integration of deployment components in the abstract behavioral specification language ABS, based on a generic cost model for resource consumption. Deployment components reflect resource-restricted executions context, and are parametric in their allocated resources. The cost model may be adapted to specific resources such as concurrent processing capacities or memory. We use our simulation tool to analyze system response time for given usage scenarios, depending on the amount of resources allocated to the deployment components and the specific cost model.

2 Deployment Components for Timed ABS

ABS is a formally defined, abstract behavioral specification language for executable object-oriented models [4]. It is designed to model distributed systems that communicate by exchanging messages. ABS consists of a functional part to define and modify user-defined datatypes, and an object-oriented, imperative part that models distributed asynchronous communication between active objects. The language has a Java-like syntax, a formal semantics given in rewriting logic, and it is executable in Maude [3]. We work with a timed extension to ABS where time evolves uniformly (see [2]), and is comparable to a system clock which updates every n milliseconds.

Deployment Components with Parametric Resources are resource-restricted execution contexts which allow us to specify and compare different execution environments for concurrent ABS models [1, 5]. Deployment components restrict the inherent concurrency of objects in ABS by mapping the logical concurrency to a model which includes a parametric number of available resources. These resources are shared between the component's objects. Resource-restricted deployment components are integrated in ABS as follows. Resources are modeled by a data type *Resource* which extends the natural numbers with an "unlimited resource" ω. Deployment components can be dynamically created with a given amount of resources r. The execution inside a deployment component is restricted by the resources currently available in the component; thus the execution in an object may need to wait for resources to become available. The availability of resources depends on the resources initially allocated to a deployment component and on the *cost model* $\mathscr{M}$ which reflects the resource usage following the execution flow of the object-oriented model.

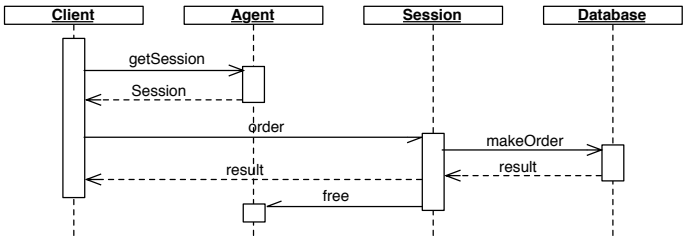


Figure 1: Example scenario

In order to give the modeler explicit control over the consumption of resources, standard statements s of ABS have associated cost, which can be specified as an expression e over the state variables of the object and the local variables of the method. If e evaluates to n in the current state, the execution of s in this state can only happen if at least n resources are available, and the execution of the statement consumes n resources from the deployment component. In a deployment component with ω resources, a statement can always be executed immediately. In a deployment component with less than n allocated resources, the statement can never be executed. Otherwise, the execution of the statement may need to wait until time has advanced in order for sufficient resources to be available.

3 Example: A Distributed Shopping Service

We show the design of a simple model of a web shop with a given time budget (the expected response time) for each client interaction. We envisage the following main sequence of events in a client session, which is shown in Figure 1. Clients connect to the shop by calling the *getSession* method of an *Agent* object, which hands out *Session* objects from a dynamically growing pool. Clients call the *order* method of their *Session* instance, which in turn calls the *makeOrder* method of a *Database* object that is shared across all sessions. The *order* call returns *True* if the order was completed within the specified time budget. After completing the order, the session object is returned to the agent's pool. This scenario models the architecture and control flow of a database-backed website, while abstracting from many details (load-balancing thread pools, data model, sessions spanning multiple requests, etc.), which can be added to the model if needed.

Figure 2 illustrates the kind of results possible to obtain after running different simulations varying in the number and behavior of clients and in the available resources; i.e., from 10 to 50 synchronous clients and 10 to 50 available resources on a deployment component. For synchronous clients, starting with 20 clients, the number of requests goes up linearly with the number of resources, indicating that the system is running at full capacity. Moreover, the number of successful requests decreases somewhat with increasing clients since communication costs also increase. For periodic clients, the system gets overloaded much more quickly since clients will have several pending requests; hence, only 2 to 10 periodic clients were simulated. It can be seen that the system becomes completely unresponsive quickly when flooded with requests.

4 Conclusions

Software modeling and analysis usually abstracts from low-level deployment aspects. As software is increasingly being developed for varying architectures, there is a need to model and to reason about deployment choices and about how a targeted architecture affects the behavior of a software

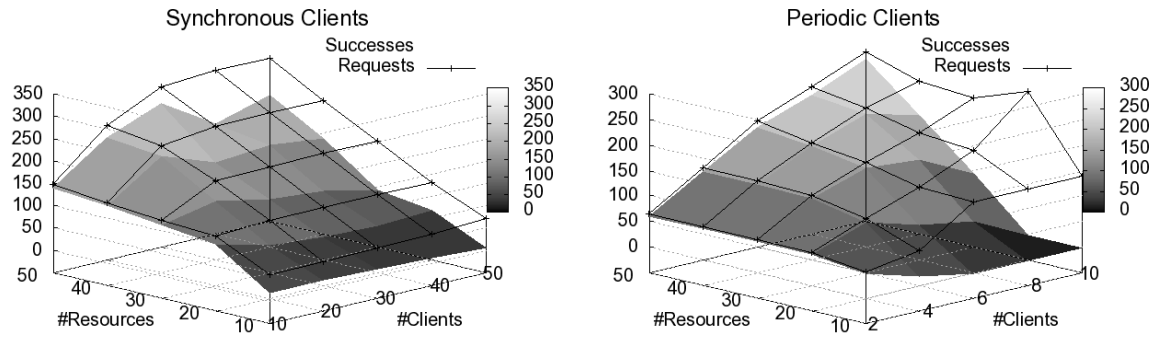


Figure 2: Number of total and successful requests, depending on the number of clients and resources, for synchronous (left) and periodic (right) clients.

system. The resources available to a software component influence the quality of service offered by the component even if the functionality is unchanged, for example with respect to response time. High-level modeling languages today do not meet this need in a satisfactory way. In order to specify and analyze the effect of deployment choices early in the software development process, these choices must be expressible at the abstraction level of the modeling language.

As a step in this direction, we propose to represent certain low-level deployment aspects in high-level modeling languages in terms of resource allocation and usage. In order to integrate resource usage into models in a natural way, it is essential that the model reflects the control flow of the system. We argue that abstract executable modeling languages offer the best abstraction level for integrating deployment aspects. Our approach is to introduce deployment components, which act as resource-restricted execution contexts for groups of concurrent objects, and are parametric in the amount of resources they make available to their objects. This makes it possible to analyze the behavior of a model ranging over the resources available in different deployment scenarios.

References

[1] E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. In M. Butler and W. Schulte, editors, *FM 2011*, LNCS 6664, pages 353–368. Springer, June 2011.

[2] J. Bjørk, E. B. Johnsen, O. Owe, and R. Schlatte. Lightweight time modeling in Timed Creol. *Electronic Proceedings in Theoretical Computer Science*, 36:67–81, 2010. *Proceedings of 1st International Workshop on Rewriting Techniques for Real-Time Systems (RTRTS 2010)*.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, LNCS 4350. Springer, 2007.

[4] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, LNCS 6957, pages 142–164. Springer, 2011. To appear.

[5] E. B. Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa. Dynamic resource reallocation between deployment components. In J. S. Dong and H. Zhu, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'10)*, LNCS 6447, pages 646–661. Springer, Nov. 2010.

[6] A. Vulgarakis and C. C. Seceleanu. Embedded systems resources: Views on modeling and analysis. In *Proc. 32nd IEEE Intl. Computer Software and Applications Conference (COMPSAC'08)*, pages 1321–1328. IEEE Computer Society, 2008.

Polymorphic behavioural lock effects for deadlock checking

Ka I Pun[1], Martin Steffen[1], Volker Stolz[1,2]
[1] Department of Informatics, University of Oslo, Norway
[2] United Nations University—Intl. Inst. for Software Technology, Macao

Deadlocks are a common problem for concurrent programs, in particular where multiple threads are accessing shared mutually exclusive resources synchronized by locks. As the scheduling at run-time affects the occurrence of a deadlock, deadlocks may only occur occasionally, and therefore are difficult to detect. Whether or not a deadlock exists in a specific run in a particular program mainly depends on if the running program encounters a number of processes forming a circular chain, where each process waits for shared resources held by the others [4].

One common way to prevent deadlocks is to statically ensure that such cycles on locks or resources in general can never occur. This can be achieved by arranging shared resources in some partial order and enforcing that the resources are accessed in accordance with that order. This idea has, e.g., been formalized in a type-theoretic setting in the form of deadlock types [3]. The static system presented in [3] supports also type *inference* (and besides deadlocks, prevents race conditions, as well). Deadlock types are also used in [1], but not for static deadlock prevention, but for improving the efficiency for deadlock avoidance at run-time.

In contrast, we use a behavioural type and effect system [2, 6] to capture lock interaction and use that behavioural description to explore an abstraction of the state space to detect potential deadlocks. The effects of our system express the relevant behaviour of a concurrent program with regard to re-entrant locks. To detect potential deadlocks, we execute the abstraction of the actual behaviour to spot cyclic waiting for shared locks among parallel threads in the program. In our previous work [7], we define a type and effect system formalizing the sketched approach. The system presented there has two important restrictions: first of all, it is *explicitly* typed, which forces the user to declare functions by specifying its expected lock behaviour (in terms of the function's effect). Putting that burden on programmers is clearly unwelcome. Secondly, based on sub-effecting as the only form of polymorphism, the formalization suffers from a lack of precision and therefore reports more spurious deadlocks than necessary. To improve the precision, we propose in this paper a *lock-polymorphic extension* , of that work, which addresses the two mentioned weaknesses. The formulation also can serve as the specification for type and effect inference system.

Parametric lock effects

We use a behavioural type and effect system to capture the interaction of shared locks. Characterizing the behaviour of each thread in a program as sequences of lock interactions allows detecting the symptom of deadlocks, i.e. waiting for shared locks in a cyclic chain.

The grammar of the effects which we use to abstractly represent the behaviour of a simple concurrent calculus with reentrant locks is presented in Fig. 1. To track which locks are actually handled in the interactions, we annotate each lock with the corresponding program point π of its creation to specify which lock is referring to at static time. As we focus on detecting deadlocks due to shared locks, we improve the precision by introducing location variables, ρ, representing lock locations. Effects can be either global in a program, or local in one single thread. For the effect construct, $\|$ represents multiple threads running in parallel globally, while semicolon represents sequential composition and + a choice among effects. The behaviour of function abstraction and recursive function is parametrized by the location variable ρ. The behaviour of lock handling: creating, locking and releasing a lock, is represented by $\nu \mathrm{L}^{r}, \mathrm{L}^{r}.\texttt{lock}$, and $\mathrm{L}^{r}.\texttt{unlock}$, respectively.

$$
\begin{array}{rcll}
\Phi & ::= & \mathbf{0} \mid p\langle\varphi\rangle \mid \Phi \parallel \Phi & \text{effects (global)} \\
\varphi & ::= & \varepsilon \mid \varphi;\varphi \mid \varphi+\varphi \mid ee(\vec{r}) \mid \alpha & \text{effects (local)} \\
ee & ::= & X \mid \lambda\vec{\rho}.\varphi \mid recX(\vec{\rho}).\varphi & \text{parametric behavior} \\
a & ::= & \texttt{spawn}\ \varphi \mid \nu \mathrm{L}^r \mid \mathrm{L}^r.\texttt{lock} \mid \mathrm{L}^r.\texttt{unlock} & \text{labels/basic effects} \\
\alpha & ::= & a \mid \tau & \text{transition labels} \\
r & ::= & \pi \mid \rho & \text{location annotations}
\end{array}
$$

Figure 1: Types and effects

A behavioural type and effect system

The type and effect system uses judgments of the form $\Gamma \vdash e : T :: \varphi$, which is read as: under the environment Γ, expression e has type T and effect φ. Three typical rules of system are sketched in Fig. 2. They deal with thread creation as well as interaction with an existing lock, where L^r represents a lock which is created at r, where r is either a program point or a location variable.

$$\frac{\Gamma \vdash e : T :: \varphi}{\Gamma \vdash \texttt{spawn}\ e : \texttt{Thread} :: \texttt{spawn}\ \varphi}\ \text{TE-SPAWN}$$

$$\frac{\Gamma \vdash v : \mathrm{L}^r :: \varphi}{\Gamma \vdash v.\,\texttt{lock} : \mathrm{L}^r :: \varphi;\mathrm{L}^r.\,\texttt{lock}}\ \text{TE-LOCK}$$

$$\frac{\Gamma \vdash v : \mathrm{L}^r :: \varphi}{\Gamma \vdash v.\,\texttt{unlock} : \mathrm{L}^r :: \varphi;\mathrm{L}^r.\texttt{unlock}}\ \text{TE-UNLOCK}$$

Figure 2: Type and Effect System

$$\sigma \vdash p_1\langle(\texttt{spawn}\ \varphi);\varphi'\rangle \xrightarrow{p\langle\texttt{spawn}\ \varphi\rangle} \sigma \vdash p_1\langle\varphi'\rangle \parallel p_2\langle\varphi\rangle \quad \text{RE-SPAWN}$$

$$\frac{\sigma(\pi) = free \vee \sigma(\pi) = p(n) \qquad \sigma' = \sigma + \pi_p}{\sigma \vdash p\langle \mathrm{L}^\pi.\,\texttt{lock}\rangle \xrightarrow{p\langle \mathrm{L}^\pi.\texttt{lock}\rangle} \sigma' \vdash p\langle\varepsilon\rangle}\ \text{RE-LOCK}$$

$$\frac{\sigma(\pi) = p(n) \quad n > 1 \quad \sigma' = \sigma - \pi_p}{\sigma \vdash p\langle \mathrm{L}^\pi.\,\texttt{unlock}\rangle \xrightarrow{p\langle \mathrm{L}^\pi.\texttt{unlock}\rangle} \sigma' \vdash p\langle\varepsilon\rangle}\ \text{RE-UNLOCK}$$

Figure 3: Operational semantics for effects

The effect system describes the behaviour of a program in terms of sequences of lock interactions among parallel processes. We detect deadlocks by executing the abstraction of the actual behaviour and spotting processes waiting for shared locks in a circular chain. The analysis of the abstract behaviour easily leads to state space explosion as different interleavings of the threads must be considered. Three rules of the operational semantics for effects corresponding to the typing rules in Fig. 2 are sketched in Fig. 3. The notation $\xrightarrow{p\langle\varphi\rangle}$ means that a step with effect φ of a thread p is executed. To tackle infinite executions through recursion which may lead to an infinite reachable state space, we place an upper bound on lock counters which are used to keep track of how often a re-entrant lock has been taken by the same thread. In addition, we bound non-tail recursive function calls by putting a similar limit on the recursion depth; for details, see [7]. Beyond that chosen limit, the behaviour is over-approximated by arbitrary, chaotic behaviour. The state space of this abstraction is finite and therefore allows exhaustive search for deadlocks. We furthermore define the notion of *deadlock and termination sensitive simulation* [5] to show that the behaviour of a program has been correctly captured in the abstraction.

Current Research Results

With the proposed specification of a type and effect system, we can automatically check for deadlocks in the five dining philosophers in around 2.5 minutes with 82269 states. Our approach correctly detects the deadlock situation in the original program without reporting any false positives. Also, our approach correctly certifies the amended version of the dining philosophers where one of the philosophers will always pick the right fork first as safe. We prove the correctness of the abstraction with regard to this

simulation of the original program, i.e., if an abstraction is deadlock free, then its original program must be deadlock free, but not vice versa: a deadlock in the abstraction, as an over-approximation, does not necessarily exist in the concrete program.

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

References

[1] R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with state analysis and run-time monitoring. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Proceedings of the Haifa Verification Conference 2005*, volume 3875 of *Lecture Notes in Computer Science*, pages 191–207. Springer-Verlag, 2006.

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

[3] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation (San Diego, California)*. ACM, June 2003.

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

[5] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann, 1971.

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

[7] K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. Technical report 404, University of Oslo, Dept. of Informatics, Mar. 2011.

A Succinct Canonical Register Automaton Model

Sofia Cassel*, Falk Howar†, Bengt Jonsson*, Maik Merten† and Bernhard Steffen†
*Dept. of Information Technology, Uppsala University, Sweden
Email: {sofia.cassel,bengt.jonsson}@it.uu.se
†Chair of Programming Systems, University of Dortmund, Germany
Email: {falk.howar,maik.merten,steffen}@cs.tu-dortmund.de

Note: Supported in part by the European FP7 project CONNECT (IST 231167). A conference version of this work will be presented at ATVA 2011: 9th International Symposium on Automated Technology for Verification and Analysis Taipei, Taiwan, October 11-14, 2011.

***Abstract*—We present a novel canonical automaton model, based on register automata, that can easily be used to specify protocol or program behavior. More concretely, register automata are reminiscent of control flow graphs: they comprise a finite control structure, assignments, and conditionals, allowing to assign values of an infinite domain to registers (variables) and to compare them for equality. A major contribution is the definition of a canonical automaton representation of any language recognizable by a deterministic register automaton, by means of a Nerode congruence. Not only is this canonical form easier to comprehend than previous proposals, but it can also be exponentially more succinct than these. Key to the canonical form is the symbolic treatment of data languages, which overcomes the structural restrictions in previous formalisms, and opens the way to new practical applications, e.g., in automata learning.**

I. Introduction

Automata models that process words or trees over infinite alphabets are becoming increasingly important in many areas, including specification, verification, and testing (e.g., [2], [22]), databases [1], and user modeling [6]. A natural form for such models consists of a finite control structure, augmented by a finite set of registers (aka state variables), processing input symbols using a predefined set of operations (tests and updates) over input data and registers. Specialized classes, such as timed automata [2], counter automata, and data-independent transition systems [18] have long been used for specification and verification. From a language-theoretic perspective, decision problems and connections with logics have been studied (e.g., [10], [8], [25]).

Modeling and reasoning with automata models can be made much more efficient if it is possible to transform models into a canonical form. Transformation into a canonical form is heavily used in verification, equivalence, and refinement checking, e.g., using (bi)simulation based criteria [17], [21]. It is the central principle underlying many techniques in automata learning (aka regular inference) that construct minimal finite automata from a finite sample of accepted and rejected words [3], [12], [23]. While for finite automata, there are standard algorithms for determinization and minimization, based on the Myhill-Nerode theorem [14], [20], it has proven difficult to carry over such constructions and define canonical forms for automata models over infinite alphabets, including timed automata [26]. Often, canonical forms are obtained at the price of (re-)encoding extensive information about the relation between parameter values in the state space (e.g., [19], [4]).

In this paper, we present a novel canonical automaton model, based on a form of register automata (RA). We define a form of RAs that are particularly suited to faithfully model a large class of systems that do not compute or manipulate data but manage their adequate distribution, e.g., protocols, as well as certain mediators and connectors. This class of systems is the backbone to support the large-scale, seamless integration and orchestration of, e.g., (Web) services to complex business applications running on the (Inter)net. One concrete current example for the application of such automata models is the CONNECT Project [15], which aims at dynamically synthesizing required connectors based on descriptions of component behavior in the form of automata.

RAs have a finite control structure. They process words over an infinite alphabet consisting of terms with parameters from an infinite domain. RAs can thus can be regarded as a simple programming language, with variables, parallel assignments, and conditions. In contrast to other types of automata that have been suggested for data languages [25], [7], our form of RAs do not restrict the access to variables to a specific order or pattern, nor do they constrain the contents of the variables (e.g., by uniqueness). This supports a much more intuitive modeling of data languages, while leaving the expressiveness untouched.

We present a Nerode congruence for RAs that yields a canonical form. Key to this generalization of Nerode's right congruence ([14], [20]) to RAs is the symbolic treatment of data languages in a way that abstracts from concrete data values and rather concentrates on the relations between parameter values. This does not only allow for the required flexibility, but it also leads to a more elegant canonical form, which may even be exponentially more succinct than other suggested canonical forms. This is very important in many applications. For instance, in automata learning, the complexity of the learning procedure directly depends on the size of the minimal canonical form of the automaton [3], [23].

By a non-technical analogy, we could compare the difference between the automata of [11], [4] and our canonical form to the difference between the region graph and zone graph

constructions for timed automata. The region graph considers all possible combinations between constraints on clock values, be they relevant to acceptance of the input word or not, whereas the zone graph construction aims to consider only relevant constraints. The analogy is not perfect, however, since our automata are always more succinct than those of [11], [4].

In summary, the contribution of this paper is a succinct and intuitive RA formalism that can easily be used to specify protocol or program behavior, with a canonical representation of any (deterministic) RA-recognizable data language by means of a Nerode congruence.

a) Related Work: Generalizations of regular languages to infinite alphabets have been studied previously. Kaminski and Francez [16] introduced finite memory automata (FMA) that recognize languages with infinite input alphabets. Since then, a number of formalisms have been suggested (pebble automata, data automata, ...) that accept different flavors of data languages (see [25], [8], [7] for an overview). Most of these formalisms recognize data languages that are invariant under permutations on the data domain. In [9] a logical characterization of data languages is given plus a transformation from logical descriptions to automata.

While most of the above mentioned work focuses on nondeterministic automata and are concerned with closedness properties and expressiveness results of data languages, we are interested in a framework for deterministic RAs that can be used to model the behavior of protocols or (restricted) programs. This includes in particular, the development of canonical models on the basis of a new Myhill Nerode-like theorem.

Only in [11], [4] a Myhill-Nerode theorem for a form of register automata is presented. Canonicity is achieved by restricting how state variables are stored, which leads to complex and hardly comprehensible models, as argued in [13]. These complications are overcome in our structurally much easier RA-based approach.

II. Outline of Main Concepts of the work

In this section, we give an outline of the technical development of the paper. We assume an unbounded domain D of data values and a set I of *actions*. A *data symbol* is a term of form $\alpha(d_1, \ldots, d_n)$, consisting of an action α and data values $d_1, \ldots, d_n$. A *data word* is a sequence of data symbols. A *data language* is a set of data words, which is closed under permutations on D.

We present an automaton model that recognizes data languages, called *Determinate Register Automata* (DRAs). Here, we illustrate it by modeling the behavior of a fragment of the XMPP protocol (shown in Figure 1). A user can register an account (providing a username and a password), log in using this account, change the password, and delete the account. We can describe this as a data language, containing sequences of data symbols $\alpha(d_1, \ldots, d_n)$ where α is an action and $d_1, \ldots, d_n$ data values. For example, the user Bob could register his account with the action register(Bob, secret) (providing his username and password), and then log in with the action login(Bob, secret). Once logged in, he could change his password to boblovesalice with the action pw(boblovesalice). In the figure, accepting locations are denoted by two concentric circles. Note that several transitions are omitted for brevity.

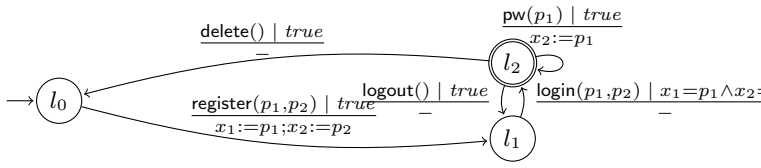


Fig. 1. Partial model for a fragment of XMPP

We omit the definitions of runs, acceptance, etc., which are not surprising.

A given data language may be accepted by many different DRAs. In order to obtain a succinct, canonical form of DRAs, we first introduce a canonical form for runs of a DRA, called *constrained words*, which capture the equalities and inequalities between parameters. Intuitively, these can be thought of as representations of runs of a register automataon. We can then use sets of constrained words, together with a classification of these words as "accepted" or "rejected", as a representation of data languages. We establish, as a central result that any data language can be represented by a *minimal* set of constrained words. This minimal set will correspond to the set of runs of our canonical automaton, and will serve several purposes during automata construction: (1) it will allow us to keep only the essential relations between data values and filter out inessential (accidental) relations between data values, (2) from it, we can derive the parameters an automaton must store in variables after processing a data word, and (3) we can transform parts of it directly into transitions when constructing the canonical DRA.

We also compare our register automata to previously proposed formalisms. There are already proposals for DRAs that accept data languages, which, however, fail to be simple and do not exactly match the flavor of data languages we are using [16], [4]. For instance, in these automata, variables have to be unique, or can only be accessed in a queue-like fashion. A Myhill-Nerode-like theorem has been proposed for these data languages and automata [11], [4]. It is, however, formulated on the level of concrete data words. This makes it difficult to encode only non-accidental relations between parameters in the corresponding canonical form.

Both the design of the DRAs and the Nerode congruence on the level of data words thus require encoding information about accidental relations between parameters into the set of locations. This makes the models harder to understand and work with. We show that in the worst case the resulting canonical models can be exponentially bigger than our canonical models.

III. Conclusions and Future Work

In this paper, we present a novel form of register automata, which also has an intuitive and succinct minimal canonical form, which can be derived from a Nerode-like right congruence.

Our immediate plans are to use these results to generalize Angluin-style active learning to data languages over infinite alphabets, which can be used to characterize protocols, services, and interfaces. Another obvious problem is to generalize the canonical model to more expressive signatures with other simple operations on data values, e.g., including comparisons of various forms.

References

[1] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: typechecking revisited. *J. Comput. Syst. Sci.*, 66(4):688–727, 2003.

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[4] M. Benedikt, C. Ley, and G. Puppis. What you must remember when processing data words. In *Proc. 4th Alberto Mendelzon Int. Workshop on Foundations of Data Management, Buenos Aires, Argentina*, volume 619 of *CEUR Workshop Proceedings*, 2010.

[5] T. Berg, B. Jonsson, and H. Raffelt. Regular inference for state machines using domains with equality tests. In *Proc. FASE*, volume 4961 of *LNCS*, pages 317–331. Springer, 2008.

[6] M. Bielecki, J. Hidders, J. Paredaens, J. Tyszkiewicz, and J. V. den Bussche. Navigating with a browser. In *Proc. ICALP '2002*, *LNCS 2380*, pp. 764–775. Springer, 2002.

[7] H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theoretical Computer Science*, 411:702–715, January 2010.

[8] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words, 2011. ACM Transactions on Computational Logic, to appear.

[9] P. Bouyer. A logical characterization of data languages. *Information Processing Letters*, 84:200–2, 2001.

[10] P. Bouyer, A. Petit, and D. Thérien. An algebraic approach to data languages and timed languages. *Information and Computation*, 182(2):137–162, 2003.

[11] N. Francez and M. Kaminski. An algebraic characterization of deterministic regular languages over infinite alphabets. *Theoretical Computer Science*, 306(1-3):155–175, 2003.

[12] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[13] O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In *Proc. LATA*, volume 6031 of *LNCS*, pages 561–572, 2010.

[14] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[15] V. Issarny, B. Steffen, B. Jonsson, G. S. Blair, P. Grace, M. Z. Kwiatkowska, R. Calinescu, P. Inverardi, M. Tivoli, A. Bertolino, and A. Sabetta. CONNECT Challenges: Towards Emergent Connectors for Eternal Networked Systems. In *ICECCS*, pages 154–161, IEEE, 2009.

[16] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[17] P. Kanellakis and S. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation*, 86(1):43–68, May 1990.

[18] R. Lazic and D. Nowak. A unifying approach to data-independence. In *Proc. CONCUR 2000*, volume 1877 of *LNCS*, pages 581–595, 2000.

[19] O. Maler and A. Pnueli. On recognizable timed languages. In *Proc. FOSSACS'04*, volume 2987 of *LNCS*, pages 348–362. Springer Verlag, 2004.

[20] A. Nerode. Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[21] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, 1987.

[22] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. on Software Engineering*, 30(1):29–42, 2004.

[23] R. Rivest and R. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, 1993.

[24] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), March, 2011.

[25] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *Proc. CSL*, volume 4207 of *LNCS*, pages 41–57. Springer, 2006.

[26] T. Wilke. Specifying timed state sequences in powerful decidable logics and timed automata. In *Proc. FTRTFT'94*, volume 863 of *LNCS*, pages 694–715. Springer, 1994.

The winning ways of concurrent games

Glynn Winskel

University of Cambridge, United Kingdom

Abstract

This talk will introduce and motivate concurrent games and winning strategies, show how winning strategies compose to yield a bicategory and how this specializes to an order-enriched category of winning deterministic concurrent strategies. A motivation has been to develop an intensional domain theory, in the spirit of game semantics, that right from the start also copes with concurrent computation. I'll try to summarize the present state of progress and relations with existing (generalized) domain theories.

Stuttering in Abstract Probabilistic Automata

Benoît Delahaye,
Aalborg University, Denmark
benoit@cs.aau.dk

Kim G. Larsen,
Aalborg University, Denmark
kgl@cs.aau.dk

Axel Legay,
INRIA/IRISA, France
axel.legay@irisa.fr

Mikkel L. Pedersen,
Aalborg University, Denmark
mikkelp@cs.aau.dk

1 Context

Nowadays, systems are tremendously big and complex and mostly result from the assembling of several components. These components are usually designed by teams working *independently* but with a common agreement on what the interface of each component should be. These interfaces precise the behaviors expected from each component as well as the environment in which they can be used, but do not impose any constraint on how the components are implemented.

Instead of relying on Word/Excel text documents or modeling languages such as UML/XML, as is usually done in practice, we recommend relying most possibly on mathematically sound formalisms. Mathematical foundations that allow to reason at the abstract level of interfaces, in order to infer properties of the global implementation, and to design or to advisedly (re)use components is a very active research area, known as *compositional reasoning* [3]. Aiming at practical applications *in fine*, the software engineering point of view naturally leads to the following requirements for a good theory of interfaces.

- **Satisfaction.** It should be decidable whether an interface admits an implementation (a model). One should also be capable of synthesizing an implementation for such an interface. In this paper, implementation shall not be viewed as a programming language but rather as a mathematical object that represents a set of programming languages sharing common properties.

- **Refinement.** It is important to be able to replace a component by another one without modifying the behaviors of the whole design. At the level of interfaces, this corresponds to the concept of *Refinement*. Refinement allows replacing, in any context, an interface by a more detailed version of it. Refinement should entail substitutability of interface implementations, meaning that every implementation satisfying a refinement also satisfies the larger interface. Refinement thus extends the classical simulation relation between systems to specifications.

- **Conjunction.** Large systems are concurrently developed for their different *aspects* or *viewpoints* by different teams using different frameworks and tools. The issue of dealing with multiple aspects or multiple viewpoints is thus essential. This implies that several interfaces are associated with a given component, namely (at least) one per viewpoint. These interfaces are to be interpreted in a conjunctive way.

- **Composition.** The interface theory should also provide a combination operation, which reflects the standard interaction/composition between systems.

In addition, any good interface theory should guarantee classical properties such as independent implementability that allows to combine components in various orders.

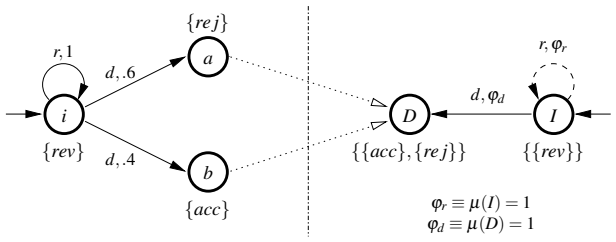


Figure 1: Implementation PA (left) and specification APA (right) of a reviewer

2 The Challenge

Building good interface theories has been the subject of intensive studies among which one finds classical logical specifications, various process algebrae such as CSP, or Input/Output automata/interfaces (see [4, 1, 6]). Recently, a new series of works has concentrated on *modal specification* [5], a language theoretic account of a fragment of the modal mu-calculus logic which admits a richer composition algebra with composition, conjunction and even residuation operators. The interest of modal automata lies in the fact that this is the only model where both composition and conjunction can be expressed and computed in a very simple and elegant manner [7].

As soon as systems include randomized algorithms, probabilistic protocols, or interact with physical environment, probabilistic models are required to reason about them. This is exacerbated by requirements for fault tolerance, when systems need to be analyzed quantitatively for the amount of failure they can tolerate, or for the delays that may appear. As Henzinger and Sifakis [3] point out, introducing probabilities into design theories allows assessing dependability of IT systems in the same manner as commonly practiced in other engineering disciplines.

Our response to this problem was to propose Constraint Markov Chains (CMC) that is a complete specification theory for pure stochastic systems, namely Markov Chains (MC). Roughly speaking, a CMC is a MC equipped with a constraint on the next-state probabilities from any state. An implementation for a CMC is thus a MC, whose next-state probability distribution satisfies the constraint associated with each state. Contrary to Interval Markov Chains where sets of distributions are represented by intervals, CMCs are closed under both composition and conjunction.

However CMCs do not permit to reason on non-deterministic behaviors, hence on Probabilistic Automata (PA). A solution to this problem was provided in [2], where we have presented Abstract Probabilistic Automata (APA), the first complete specification theory for probabilistic automata. APAs are specifications that represents a possibly infinite set of PAs. APAs combine Modal Automata and CMCs – the abstractions for labelled transition systems and Markov Chains, respectively. The theory has been implemented and tested on several case studies.

Example 1. *Consider the implementation (left) and specification (right) of a reviewer given in Figure 1. The specification specifies that there are two possible transitions from initial state I: a may transition with action r (read) and a must transition with action d (decide). May transitions are represented with dashed arrow, while must transitions are represented with plain arrow. The probability distributions associated with these actions are specified by the constraints φ_r and φ_d, respectively. One can see that the implementation gives a more precise behavior of the reviewer: action r loops back to initial state i with probability* 1*, while the decision leads to state a (reject) with probability* .6 *and to state b (accept) with probability* .4*. Satisfaction between implementation and specification lifts the classical notion of simulation for PA to APA as follows: (1) all must transitions of the specification must be matched with transitions in the implementations, and (2) all transitions in the implementation must be matched with may transitions in the specification. Additionally, we have to check that the probability distributions in*

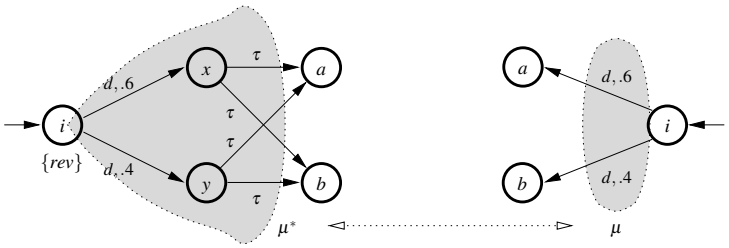


Figure 2: Illustration of weak bisimulation between PAs

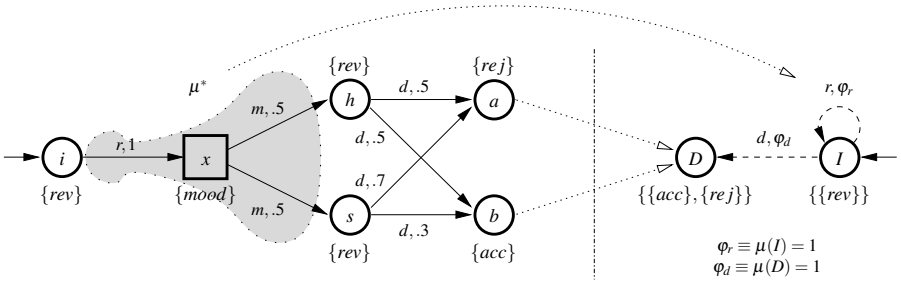


Figure 3: Illustration of stutter satisfaction for APAs

the implementation are matched with probability distributions in the specification that satisfy the given constraints.

However, while APA is a complete specification theory in itself, it still lacks some important design features that are needed to make this theory attractive from an engineering point of view. As an example, in the process of incremental design, it is sometimes necessary to incrementally widen the scope of implementations. Usually, for PA, the latter is done by permitting the addition of hidden actions also called stutter steps. Such actions clearly complicate the definition and the computation of operations such as bisimulation/simulation and composition. As an example, simulation between PAs has to be lifted to weak simulation as illustrated with the following example.

Example 2. *In Figure 2, we illustrate the notion of weak bisimulations between two given PAs. In the left PA, internal action τ has to be taken into account. In this case, one has to compute the overall probability μ^* of reaching states a and b from state i and compare this probability to the transition probability μ in the right PA. In essence, weak bisimulation holds if these probabilities are equal.*

The objectives of this presentation are to survey the theory of APAs as well as to present a solution to the treatment of hidden actions. Our solution takes the form of an extension of the one proposed to handle internal actions of PAs: we say that a distribution μ^* is reached via a stutter transition a if there exists a scheduler for the internal transitions that can follow the action a, such that the overal distribution reached after executing only internal actions is μ^*. The idea is illustrated with the following example.

Example 3. *In our formalism, hidden actions and local states are made explicit. As illustrated in Figure 3 the state space of PAs is split into visible (circles) and local (boxes) states. Invisible states can only perform hidden actions. Stuttering satisfaction then computes all internal overall probabilities of reaching visible states, and verifies that these overall probabilities match distributions in the specification satisfying the constraints.*

During the presentation, we shall show how extending APA with internal actions impacts both the definition and the algorithmic treatment of their operations and the properties they guarantee. Finally, we will also present a new logical characterization of APAs that takes internal actions into account.

References

[1] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proc. 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC / SIGSOFT FSE), Vienna, Austria*, pages 109–120. ACM Press, 2001.

[2] Benoît Delahaye, Joost-Pieter Katoen, Kim G. Larsen, Axel Legay, Mikkel L. Pedersen, Falak Sher, and Andrzej Wasowski. Abstract probabilistic automata. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 324–339. Springer, 2011.

[3] T. A. Henzinger and J. Sifakis. The embedded systems design challenge. In *Proc. 14th International Symposium on Formal Methods (FM), Hamilton, Canada*, volume 4085 of *lncs*, pages 1–15. Springer, 2006.

[4] H. Hermanns, U. Herzog, and J-P. Katoen. Process algebra for performance evaluation. *Theor. Comput. Sci.*, 274(1-2):43–87, 2002.

[5] K. G. Larsen. Modal specifications. In *Proc. International Workshop on Automatic Verification Methods for Finite State Systems (AVMS), Grenoble, France*, volume 407 of *Lecture Notes in Computer Science*, pages 232–246. Springer, 1989.

[6] Nancy Lynch and Mark R. Tuttle. An introduction to Input/Output automata. *CWI-quarterly*, 2(3), 1989.

[7] Jean-Baptiste Raclet. *Quotient de spécifications pour la réutilisation de composants*. PhD thesis, Université de Rennes I, december 2007. (In French).

Towards quantitative evaluation of stochastic pharmacy workflows

Luke Herbert
Technical University of Denmark
Lyngby, Denmark
lthhe@imm.dtu.dk

Robin Sharp
Technical University of Denmark
Lyngby, Denmark
robin@imm.dtu.dk

1 Introduction

European hospitals are being placed under growing pressure to deliver efficiency gains due to an increasing elderly population and ever tighter financial constraints. A significant proportion of the treatment performed relies on the adaptive application of various intravenous medicine which is mostly in liquid form and which is prepared in central pharmacies serving a number of departments. Due to strict safety requirements within pharmacies, the process of preparation of fluid medicine for intravenous use is; labour intensive, inefficient[3], error-prone[1, 6] and poses health risks to medical staff. Current product preparation workflows rely on production techniques largely abandoned by other industries; including duplicating inventory, inflexible batch processing, and over production where doses are often prepared in anticipation of a need or demand that may end up not being realised which contributes to waste[7].

A solution to these issues has been sought in the automation of the medication management and dispensing processes. Deployment of systems for this purpose has delivered significant improvements in safety for both staff and patients[2, 10], and new pharmacy capabilities such as customised medicine have become feasible[2]. However, the technology has proven disruptive to traditional pharmacy processes and many potential efficiency gains have yet to be realised.

1.1 Contribution

In this paper, we present the first steps towards addressing these issues by developing a method to model and analyse pharmacy and automated workflows and their interaction so as to be able to accurately provision an effective automated solution. This is achieved by extending a well developed modelling formalism (BPMN)[5], in use in healthcare, to include timing and stochastic data (section 2). We present an algorithm for the conversion of this formalism into a format amenable to *stochastic model checking*, which allows for the calculation of a wide range of system properties (section 3). This work will serve as a basis for the development of software tools implementing the method and further theoretical work aimed at automatic synthesis of process configurations (section 4).

2 BPMN

Business Process Model and Notation (BPMN)[5] is a graphical notation for specifying business processes. The primary goal of BPMN is to provide a notation that is readily understandable by all business users. BPMN's ability to serve as a standardized bridge between business process design and implementation has lead to widespread adoption in the healthcare industry[16, 14] where its design goals have allowed for precise description of hospital workflows. Specific case studies[13, 15, 12] have underlined this, developing models of various complex hospital workflows rapidly and allowing effective manual restructuring of process flows.

2.1 Business Process Diagrams (BPD)

Modelling a workflow in BPMN involves composing a number of BPMN elements into a single *business process diagram* (BPD). For the purposes of this extended abstract we will consider BPDs restricted to a minimal subset of BPMN elements with two extensions to allow for timing and stochastic branching. Although this omits traditional data-based control flow, it is sufficient to illustrate a method to perform performance analysis of the resulting models using techniques from stochastic model checking.

Definition 1 (Minimal BPD)**.** *A minimal BPD is a tuple* $\mathcal{BPD} = (\mathbf{O}, \mathcal{F})$ *where* $\mathbf{O}$ *is a set of nodes (corresponding to BPMN objects) and* $\mathcal{F} \subseteq \mathbf{O} \times \mathbf{O}$ *is an edge relation (corresponding to BPMN flows). The set* $\mathbf{O}$ *can be partitioned into 7 disjoint subsets;* start *events* $\mathbf{E^S}$*,* end *events* $\mathbf{E^E}$*,* trigger *events* $\mathbf{E^T}$*, catch events* $\mathbf{E^C}$*,* $\mathbf{G^B}$ *a set of branch gateways where each element* $g \in \mathbf{G^B}$ *has an associated* rate distribution function R_g*,* $\mathbf{G^J}$ *a set of join gateways which merge control paths and* $\mathbf{T}$ *a set of tasks, where each element* $t \in \mathbf{T}$ *has an associated stochastic time delay expressed as a normal distribution function* $N_t(\mu_t, \sigma_t^2)$*.*

A $\mathcal{BPD}$ describes a number of business processes, where for each process $\mathcal{F}$ defines a directed graph with nodes which are elements of $\mathbf{O}$. For each node $o \in \mathbf{O}$ the input nodes of o are $\mathsf{in}(o) = \{x \in \mathbf{O} | x \mathcal{F} y\}$ and output nodes of o are $\mathsf{out}(o) = \{y \in \mathbf{O} | x \mathcal{F} y\}$. Branch gateways assign a rate to each outflow indicating how often the flow of control takes this path and thus controlling execution. This definition of *BPDs* allows for graphs which are unconnected, do not have start or end elements or various other properties which place them outside what is permitted in BPMN models. We therefore define:

Definition 2 (Well-formed minimal BPD)**.** *A* $\mathcal{BPD}$ *is well-formed if the following conditions hold:*

$$
\begin{array}{ll}
\textbf{\textit{WF1:}}\ \forall e \in \mathbf{E^S} : \mathsf{in}(e) = \varnothing \wedge |\mathsf{out}(e)| = 1 & \textbf{\textit{WF2:}}\ \forall e \in \mathbf{E^E} : \mathsf{out}(e) = \varnothing \wedge |\mathsf{in}(e)| = 1 \\
\textbf{\textit{WF3:}}\ \forall t \in \mathbf{T} : |\mathsf{out}(t)| = 1 \wedge |\mathsf{in}(t)| = 1 & \textbf{\textit{WF4:}}\ \forall g \in \mathbf{G^B} : |\mathsf{in}(g)| = 1 \wedge |\mathsf{out}(g)| > 1) \\
\textbf{\textit{WF5:}}\ \forall g \in \mathbf{G^B} : \sum_{\forall x_i \in \mathsf{out}(g)} R_g(x_i) = 1 & \textbf{\textit{WF6:}}\ \forall g \in \mathbf{G^J} : |\mathsf{in}(g)| > 1 \wedge |\mathsf{out}(g)| = 1) \\
\textbf{\textit{WF7:}}\ \forall o \in \mathbf{O}, \exists (s, e) \in \mathbf{E^S} \times \mathbf{E^E} : s \mathcal{F}^* o \wedge o \mathcal{F}^* e &
\end{array}
$$

where $\mathcal{F}^*$ *is the reflexive transitive closure of* $\mathcal{F}$*.*

The first two conditions **WF1** and **WF2** simply state that start and end states do not have respectively in or out flows and are followed/preceded by a single state. **WF3** ensures tasks do not branch control flow. **WF4 WF6** ensure that split and join of control flows actually split/join flow. **WF5** ensures that the rates of all branches are defined and hence no branch gateway can be reached from which a further choice is not possible. Finally **WF7** ensures that all objects lie on a path from a start to an end event. We will only consider *well-formed minimal BPDs*. However, it should be noted that this language has features that cover the vast majority of the core BPMN constructs with only data based control flow being absent. Many elements of data-based control flow can, however, be simulated in well-formed minimal BPDs using message passing and dummy processes.

2.2 BPMN Semantics

BPMN is a visual notation and while the BPMN specification[5] provides extensive syntactic rules, the semantics of BPMN is only given in narrative form using a somewhat inconsistent terminology. A number of papers have undertaken the task of providing formal semantics in the form of Petri nets[4], Business Process Execution Language (BPEL),[11] and Communicating Sequential Processes (CSP) [17].

In the work introduced in this paper we adopt the method for deriving a CSP semantics for a BPMN fragment given in [17] to our extended subset of BPMN. The basic idea of determining the semantics of BPMN is as follows: an abstracted BPMN syntax is expressed in the Z notation and then a semantic function converts this to a parallel composition of CSP processes corresponding to states in the diagram. These processes are themselves built up from smaller predefined CSP processes used as building blocks. This development is verbose but quite straightforward and can be used in our case without dramatic modification other than accommodating timing, in the form of a single global clock, and a slight modification of non-deterministic choice to accommodate stochastic rates.

3 Stochastic Model Checking

The main goal with this work is to be able to perform stochastic model checking of BPMN models. Specifically we wish to derive properties of the form:

- **Transient and steady-state probabilities** e.g. the probability that the system operational at time instant t or the overall probability that it is operational.
- **Timing, occurrence and ordering of events** e.g. the probability that a failure of component *B* (if it occurs) happens before any failure of component *A*.

- **Reward-based properties** e.g. the throughput of the system, i.e. the expected steady-state rate of job completion.
- **Best- and worst-case scenarios** e.g. the best-case instantaneous availability of the system at time t, starting from any initial configuration.

3.1 Conversion

Conversion of *well-formed minimal BPDs* to a specific model checking format follows broadly the process used for a wide range of model checkers as all require a formal language input. In this case we will outline our conversion to the PRISM language format[8].

We begin by decomposing a BPD into processes. We then traverse each process from its unique start node to various end nodes building a graph-like data structure in a fashion similar to[11]. Branching in the simple cases presented here maps directly to the PRISM language.

We deal with tasks $t \in \mathbf{T}$ by creating two states for a task: one before and one after the task. The first state functions as a branching gateway with a number of edges generated to the second sate which functions as a join gateway. The number of edges generated can be chosen during the conversion process by dividing the distribution into the required number of intervals, each edge has a time delay equal to the centre of the interval and a branch probability given as $p = Pr[a < X < b]$ where a and b are the bounds of the interval. (see figure 1)

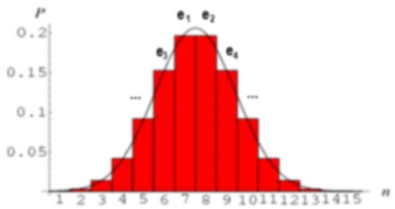


Figure 1: Task edge generation

It should be noted that our conversion process allows for a great deal of tuning, making it possible to produce models of varying complexity and with a wide range of annotations.

4 Conclusions

In this brief introduction we have extended the BPMN modelling formalism, widely used in the modelling of business operations, to allow for the recording of variable task timings and stochastically branching control flow. We have outlined a means to determine the semantics of such models and outlined how these models can be converted to a format suitable for verification by model checking. This abstract omits many details of the methods being developed and seeks to demonstrate results for a very limited subset of BPMN. It should be stressed that a fuller presentation of this work would present a complete semantics for a larger timed stochastic BPMN fragment, which consequently also would allow for a more extensive description of the method of conversion to a model checkable system description. This work is ongoing, and the theoretical developments described are accompanied by the development of software tools making use of the PRISM model checker[9].

4.1 Future work

The ultimate goal of this work is to investigate the following types of synthesis problems within the context given.

- Given a timed BPMN workflow, including probabilistic branching, calculate the next sequence of actions to be taken for all states in the systems to obtain the optimum of some parameter. (E.g. Given a medical robot interacting with pharmacists required to perform a given list of tasks in a minimum amount of time, what action should it take at each stage depending on the outcome of its own, unpredictable, operations.).
- Given a timed BPMN workflow, and a new process to be added to this workflow calculate the optimal configuration of the new combined workflow, with respect to some property of system, such as time. (E.g. How would be it best to reorganise a pharmacy workflow once a robotic system was introduced).
- Given an existing timed BPMN workflow and a multi-set of new processes that may be added, find the optimal choice from this multi-set to achieve the smallest/largest value of a parameter of interest. (E.g. Determine what selection of robotic sub-modules interacting with an existing workflow will achieve a reduction in drug production time/cost).

References

[1] C. A. Bond, C. L. Raehl, and T. Franke. Medication errors in united states hospitals. *The Journal of Human Pharmacology and Drug Therapy*, 21(9):1023–1036, September 2001.

[2] J. Carmenates and M. R. Keith. Impact of automation on pharmacist interventions and medication errors in a correctional health care system. *American Journal of Health-System Pharmacy*, 59(9):779–783, May 2001.

[3] A. Colquhoun. Could automation improve efficieny and help pharmacies with cost saving? *The Pharmaceutical Journal*, 285:587–591, November 2010.

[4] R. M. Dijkman, M. Dumas, and C. Ouyang. Formal semantics and analysis of bpmn process models. 2007.

[5] O. M. Group. *Business Process Model and Notation (BPMN) 2.0*. Object Management Group, Needham MA, USA, 2011.

[6] P. Y. Han, I. D. Coombes, and B. Green. Factors predictive of intravenous fluid administration errors in australian surgical care wards. *Quality and safety in health care*, 14:179–184, 2004.

[7] B. L. Hintzen, S. J. Knoer, C. J. V. Dyke, and B. S. Milavitz. Effect of lean process improvement techniques on a university hospital inpatient pharmacy. *American Journal of Health-System Pharmacy*, 66(22), November.

[8] M. Kwiatkowska, G. Norman, and D. Parker. Prism: probabilistic model checking for performance and reliability analysis. *SIGMETRICS Perform. Eval. Rev.*, 36:40–45, March 2009.

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

[10] S. Oswald and R. Caldwell. Dispensing error rate after implementation of an automated pharmacy carousel system. *American Journal of Health-System Pharmacy*, 64(13):1427–1431, July 2007.

[11] C. Ouyang, M. Dumas, and A. H. M. T. Hofstede. Pattern-based translation of bpmn process models to bpel web services. *International Journal of Web Services Research (JWSR)*, 5(1):42–62, 2007.

[12] J. Puustjärvi and L. Puustjärvi. Automating the coordination of electronic prescription processes. In *2006 8th International Conference on e-Health Networking, Applications and Services (Healthcom 2006)*, pages 147–151, August 2006.

[13] J. Puustjärvi and L. Puustjärvi. Automating the dissemination of information entities to healthcare professionals. In B. Papasratorn, W. Chutimaskul, K. Porkaew, and V. Vanijja, editors, *Advances in Information Technology*, volume 55 of *Communications in Computer and Information Science*, pages 123–132. Springer Berlin Heidelberg, 2009.

[14] A. A. Rad, M. Benyoucef, C. E. Kuziemsky, and A. A. Rad. An evaluation framework for business process modeling languages in healthcare. *J. Theor. Appl. Electron. Commer. Res.*, 4:1–19, August 2009.

[15] M. G. Rojo, E. Rolon, L. Calahorra, F. O. Garcia, R. P. Sanchez, F. Ruiz, N. Ballester, M. Armenteros, T. Rodriguez, and R. M. Espartero. Implementation of the business process modelling notation (bpmn) in the modelling of anatomic pathology processes. In *Proceedings of the 9th European Congress on Telepathology and 3rd International Congress on Virtual Microscopy*, volume 3 (Suppl 1), London, UK, July 2008. BioMed Central Ltd.

[16] E. Rolón, F. García, F. Ruíz, M. Piattini, and L. Calahorra. Healthcare process development with bpmn. In S. R. Cruz-Cunha M. M., Tavares A. J., editor, *Handbook of Research on Developments in E-Health and Telemedicine: Technological and Social Perspectives*, pages 1024–1047. Facultad de Ingeniería, Universidad de Talca, Talca, Chile, 2010.

[17] P. Y. Wong and J. Gibbons. A process semantics for bpmn. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM '08, pages 355–374, Berlin, Heidelberg, 2008. Springer-Verlag.

Correctness of Constraint-Aware Model Transformations

Xiaoliang Wang, Yngve Lamo
Bergen University College
{xwa,yla}@hib.no

Model transformations are important in Model Driven Engineering (MDE). They automate software development steps and greatly improve productivity and reduce software errors. However, the design of model transformation rules requires lots of manual work. To fully take advantage of MDE, correctness of model transformation rules should be ensured. In this paper, we present an ongoing work to use model checking techniques to validate model transformation rules. The work also studies how rule application strategies affect correctness and efficiency of model transformations.

1 Introduction

Model driven Engineering (MDE) turns out to be a promising software development methodology. MDE promotes the use of models as the primary artefacts in software development. Models are used to specify, simulate, generate code and maintain the resulting applications. They are manipulated by model transformations throughout the software development life cycle. In that way, consistence between models is assured and productivity is greatly improved.

In MDE, models are described in modelling languages. The most prevailing modelling language is the Unified Modelling Language (UML) [4] proposed by OMG [4]. The syntax of a modelling language is usually specified by a metamodel. For example, the metamodel of a UML model is also specified by UML. In that way, a UML model is an instance of and conforms to UML. Besides, constraints are also required to define the properties that a valid instance of the model should possess. UML uses a text-based language, the Object Constraint Language (OCL) [4] to define software constraints. Any valid UML model must satisfy the given OCL constraints.

1.1 Model Transformations

A model transformation is the automatic generation of target models from source models. The relation between source model elements and target model elements is usually specified by model transformation rules. Given a source model, a target model is expected to be constructed by applying a sequence of rules. To define the rules, a language that coordinates both the source metamodel and the target metamodel is needed. OMG has invented the Query/View/Transformation (QVT) language [4] that has become the de facto industry standard model transformation language.

Model transformations play an important role in MDE. Models are modified by model transformations for different development purposes. For example, when deploying an application, a Platform Independent Model (PIM) should be transformed into a suitable Platform Specific Model (PSM) in accordance to the hardware requirements. These models should be used for automatic code generation.

However, right now, model transformation rules are designed manually. In order to ensure reliability, it is necessary to check the correctness of the model transformation. A match of a rule in a model means a graph homomorphism can be found from the left hand side of the rule to the model. If a match of a rule is found in a model, we say that the rule is applicable to the model. A correct model transformation means that for any valid source model, a sequence of applicable rules which constructs a valid target model can be found.

Besides, the choice of which rule to apply also influences what kind of target model is constructed out of the source model. The rule application strategy controls which rule should be applied when several rules are applicable at the same time. It also decides the resulting target model when several matches

of a rule are found in the model. So the rule application strategies are also important when checking correctness.

There are existing projects like Henshin [7] that use model checking techniques to check correctness of model transformations. But so far we are not aware of any projects that uses model checking for constraint-aware models transformations. This work presents a model checking approach to validate constraint-aware model transformations. Moreover, application strategies are also studied to analyse their impact to the correctness of model transformations. The proposed approach is based on the Diagram Predicate Framework (DPF) [5] which provides a formalization of (meta)modelling and model transformation based on graph theory [2] and category theory [1].

2 Diagram Predicate Framework

DPF aims to build a fully *diagrammatic specification framework* for MDE. That is to develop and use a *diagrammatic formalism* to define and reason about models and *model transformations*. In DPF, models are formalized as diagrammatic specifications which consist of an underlying graph structure together with a set of atomic constraints. A modelling language is formalized as a modelling formalism $(\Sigma_2 \rhd S_2, S_2, \Sigma_3)$. The specification S_2 represents the metamodel of the language; the signature Σ_3 contains predicates which are used to add constraints to the metamodel S_2; while the typed signature $\Sigma_2 \rhd S_2$ contains predicates which are used to add constraints to the specification S_1 that are specified by the modelling formalism. DPF also takes constraints in model transformations [6] into account. Based on this, given a modelling formalism $(\Sigma_2 \rhd S_2, S_2, \Sigma_3)$, constraint-aware rules are formalized as a typed specification morphism $r : \mathfrak{L} \rhd S_2 \hookrightarrow \mathfrak{R} \rhd S_2$. Usually, the modelling formalism used in model transformation is a joint one which relates source and target modelling languages together.

2.1 Correctness of Model Transformations

Software programs need validation before deployment. Testing is enough for less critical applications. For more critical applications, reliability could be ensured by use of theorem provers or model checkers. Testing can never completely identify all the defects, but it can help the programmer to find bugs and refine the program. The use of theorem provers and model checkers can guarantee programs without bugs. But the application of theorem provers needs a mathematical formalization of the program and involves human activities whereas model checkers have the state explosion problem.

In a similar way validation of model transformations is also required. Model transformations are executed automatically, hence it could be possible to run automatic tests of model transformations. Hopefully, if the test result is true, a sequence of applicable rules will be given that constructs a desired target model. Otherwise, it will give feedbacks assisting the designers to correct the rules. However, there is a difference of testing programs and transformations. For any determinstic program, each input only have one execution path. For a model transformation, there maybe several different sequences of applicable rules. To validate correctness of a model transformation, all the possible sequences should be checked by a model checker. By considering models as states and rules as transitions, a model transformation can be translated into a finite state machine. In order to test the transformation rules by use of model checking, the source model and transformation rules should be translated into a Kripke structure. Before continuing, a short introduction about model checking is given.

2.2 Model checking

Model checking is an automatic way to verify that a model satisfies a given specification. Usually, the model is represented by a Kripke structure, while the specification is formalized in temporal logic, e.g. CTL or LTL [3]. Formally a Kripke structure is defined as a 4-tuple, $M = (S, I, R, L)$, where S is a finite set of states s. I is a set of initial states $I \subseteq S$. R is a transition relation $R \subseteq S \times S$ such that R is

left-total, i.e., $\forall s \in S, \exists s' \in S$, such that $(s, s') \in R$. And L is a labelling function $L : S \to 2^{AP}$, where AP is a set of *atomic propositions* [3].

Recall that the joint modelling formalism (JMF) includes the source metamodel (SMM) and the target metamodel (TMM). We now give the translation procedure from DPF model transformations to Kripke structures that are used in model checking. From the joint modelling formalism (JMF), the model transformation rules (MTRs) and the source model(SM), we construct a Kripke structure in the following way:

- We define a initial state i representing SM
- For each state $s \in S$ and for every MTR $r : \mathfrak{L} \rhd S_2 \hookrightarrow \mathfrak{R} \rhd S_2$ we check $IsMatch(Model, \mathfrak{L} \rhd S_2)$. If it is true, the rule is applicable
- For each state $s \in S$ and for every applicable MTR $r : \mathfrak{L} \rhd S_2 \hookrightarrow \mathfrak{R} \rhd S_2$, we define a new state $r(s) \in S$ and a transition $t : s \to r(s)$

Note that $IsMatch$ checks if a match of the left input pattern is found in $Model$. If so, a new state $r(s)$ and a new transition $t : s \to r(s)$ are created. An important factor here is the rule application strategy. Two situations are under consideration. Firstly, several matches may be found in a model for one rule. Secondly, several rules may be applicable to the current model. Different strategies to control these scenarioes results in different target models. Existing strategies are e.g. negative application conditions (NAC) and layering of rules. Analysis of affection of those strategies can be carried out during the model checking. Moreover, transformations are constraint-aware in DPF. That means when constructing the state space and the transition relationship the related constraints should be considered.

Following this procedure a state space can be derived. It contains all the reachable states, that are models derived by application of rules and valid instances of the JMF. The property to be checked is that in the future there is a state where no more rule is applicable and from this state a valid target model can be derived. In CTL, this property can be formalized as:

$$EF!AnyRuleApplicable(Model, MTRs)\&\&IsInstanceof(getTargetModel(Model), TMM)$$

Existing model checking technologies can be used to check if the property is satisfied. Different instances of the source metamodel can be created to test the MTRs. A valid sequence of applicable rules are obtained if the result is true. Otherwise feedback will be given to help the designer to modify the model transformation. As said before, testing can never guarantee transformation correctness, in the future we will study how theorem provers can be used to ensure correctness of model transformations.

References

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

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

[3] E. C. Jr., O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[4] Object Management Group. *Web site*. http://www.omg.org.

[5] A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

[6] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A Formalisation of Constraint-Aware Model Transformations. In D. Rosenblum and G. Taentzer, editors, *FASE 2010*, volume 6013 of *LNCS*, pages 13–28. Springer, 2010.

[7] The EMF Henshin Transformation Tool. *Project Web Site*. http://www.eclipse.org/modeling/emft/henshin/.

An Architecture-based Verification Technique for AADL-specifications

Andreas Johnsen
Mälardalen University
Västerås, Sweden
andreas.johnsen@mdhlse

Paul Pettersson
Mälardalen University
Västerås, Sweden
paul.pettersson@mdh.se

Kristina Lundqvist
Mälardalen University
Västerås, Sweden
kristina.lundqvist@mdh.se

1 Background

The architecture design phase is one of the most critical phases in the development process of software-intensive systems. The architecture specification is the initial development artefact representing the earliest design decisions made on the intended system's structure, functional properties and quality attributes (non-functional properties). Design decisions involve the allocation of functional properties – which are closely related to a system's behavior, capabilities and services – to certain structures to achieve certain quality attributes. Furthermore, the architecture specification is used as a mutual communication blueprint among stakeholders and guides the implementation phase of the system. Consequently, the developed system will heavily depend on the architecture specification, which it ideally should conform to.

The design decisions established in the architecture design phase, or the absence of some, may impose incorrect properties of the system and thereby creating challenges in quality assurance processes. These incorrect structural, functional as well as non-functional properties may go unnoticed until later phases of the development process where a correction is known to be significantly more costly compared to a correction in the architecture design phase. Hence, evaluating the architecture specification is crucial in order to detect possible faults and inconsistencies before the development process progresses, reducing a significant amount of cost and time. Furthermore, in order to preserve the valuable effort made at the architecture design phase, an implementation of the system must be implemented in conformance with the architecture specification. The verification techniques used to tackle these challenges, i.e. *1) to evaluate the completeness and the consistency of an architecture specification* and *2) to test the conformance of an implementation with respect to its architecture specification*, rely on what kind of properties and their relations that may be described with the Architecture Description Language (ADL) used to describe the system's architecture.

Software-intensive systems are systems where software interacts with sensors, actuators, devices, other systems and people. Examples of such systems are embedded systems for aerospace and automotive. What these systems have in common is that they often are operating in dynamic, time- and safety-critical environments. One ADL that has been developed for this kind of systems, is the Architecture Analysis and Design Language (AADL) [2], which is widely used both within industry and the research community.

2 Contributions

In this paper we propose an architecture-based verification technique, for software-intensive systems specified by AADL, addressing challenge *1)* and *2)* mentioned above. The technique is based on formal constructs enabling automation of the verification activities where challenge *1)* and *2)* are tackled by adapting model-checking and model-based testing approaches - respectively - to an architectural perspective. The objective of the technique is to evaluate the integration of components at both the specification-level and the implementation-level through control-flow and data-flow analysis. The analysis is performed upon the dynamic semantics of AADL to not only verify that functional properties of control and data interactions among component interfaces are correct, but also the non-functional properties such as performance, schedulability and safety. Automated checking of AADL specifications is

not feasible directly from the artefact since AADL lacks formal semantics and implemented semantics. We define transformation rules (mappings) to a network model of timed automata in Uppaal [1], which provide a formal and implemented semantics of AADL.

3 Method

This section presents an overview of the automatable verification technique for AADL specifications. The technique comprises both evaluation of specifications and the systems' conformity to them. It is depicted as a flowchart in Figure 1, where initially a system's intended architecture is specified using AADL. Such an artefact is commonly specified through a translation from something cognitive, an idea, a need or an informal/semi-formal requirement specification, but since it is informal, it is not possible to formally prove that the AADL specification correctly conforms to the informal one it is derived from. Consequently, making this type of evaluation far from possible to automate and thus is out of scope in this technique. What is possible though is to formally reason about a system solely through the AADL specification (note that requirements or decompositions of these can be formalized in AADL), to prove its consistency and completeness, and later use it as a test model to perform model-based testing on. The different steps of the verification technique are as follows:

The first step is to use the mappings/transformation rules to transform an AADL specification to a timed automata model upon which automated formal verification can be performed.

The second step is to apply the architecture-based verification criteria to the AADL specification. They define the test selection, i.e., what samples of the specification to evaluate and how they are extracted, and the coverage requirement, i.e., how many samples to evaluate. The samples generated from the criteria are sequences of component-integrations in terms of control-flows and data-flows.

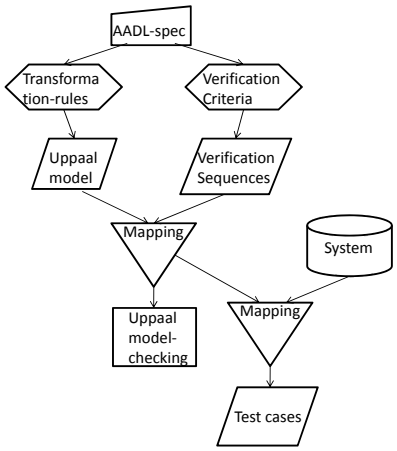


Figure 1: Flowchart of the technique

Sequences are transformed, in the third step, to the corresponding timed automata paths through a structural mapping between them.

The outcome, a set of timed automata paths are required in the fourth step to be fully checked by the Uppaal model-checker, by using temporal logics, in order to satisfy the criteria. The verdict from Uppaal reveals the consistency and completeness of the AADL specification, where a correction of the specification should be made if it is shown inconsistent or incomplete.

The paths are later used in the fifth step to generate test cases to the implementation (model-based testing), to test the conformance of the implementation with respect to the architecture specification. Test paths are transformed to concrete test cases through a mapping between the architecture specification and its implementation (we assume identical name spaces between the AADL specification and the system).

3.1 Verification Criteria

Four different types of AADL connections (*port-*, *data access-*, *subprogram call* and *parameter-connections*) represent the architectural control-flows and data-flows of an AADL specification. Architectural control-flows are the different execution orders of architectural elements whereas architectural data-flows are the relations between definitions of data elements in a source component and uses of the corresponding data elements in a target component. These flows may be dependent on *mode state machines* (represent different runtime configurations), refined by *behavioral models* (represent logical execution of threads and callable subprograms) and constrained by associated functional as well as non-functional *properties*

where conflicts may occur between these constructs. The objective of the verification criteria is to ensure consistency and completeness of and between the flows, their dependencies, their refinements and their constraints through analysis of control-flow reachability, data-flow reachability and concurrency among flows (note that these properties subsumes analysis of numerous other properties such as performance and schedulability):

Control-flow reachability: Every architectural element in an execution order should be able to reach the subsequent element to be executed in the order. The subsequent element should be reached without conflicting properties (constraints) of the execution order.

Data-flow reachability: Every data element should be able to reach its target component, where the data is used, from its source component, where the data is defined. The target component should be reached without conflicting properties of the data flow.

Concurrency among flows: Analysis of single interactions of data or control is not enough since there are implicit relations between them that may cause deadlocks in the system. The relations between the flows should not prevent control-flow reachability or data-flow reachability, and where the system should be free from deadlocks.

Analysis of these properties is generated from the five different types of AADL-relations we formally have specified. They define the possible atomic bindings of control and data in AADL, and are used to generate the control- and data-flow graphs. Based on these relations, three types of AADL-sequences are specified. They define the possible paths in the control- and data-flow graphs, and are used to generate the verification sequences. Finally, specification of three types of coverage criteria define the coverage required to verdict completeness, consistency and conformance.

3.2 Transformation to Uppaal

The dynamic semantics of AADL constructs are to a large extent dependent on the AADL execution model which defines the dynamic semantics. It consists of several different aspects of a run-time environment, such as synchronous interactions, asynchronous interactions, nominal execution, recovery execution, etc. The AADL standard specifies a default runt-time environment with synchronous interactions and preemptive scheduling. Hence, the default model (that is, the default values of *property* annotations) consists of periodic threads communicating through their data ports. As an initial effort to specify dynamic semantics of AADL constructs, we restrict the definition of transformation rules and solely consider synchronous interactions with preemptive scheduling (non-preemptive scheduling is subsumed).

4 Results and Conclusions

The AADL language is a formalism for development of safety-critical software-intensive systems. The technique evaluates the consistency and completeness of an AADL specification and tests a systems' conformity to it. The entire development process is covered by adapting a combination of model-checking and model-based testing approaches to an architectural perspective. The adaption is performed through the definition of AADL-specific verification criteria. We are currently validating the technique against a system developed by a major vehicle manufacturer.

References

[1] Gerd Behrmann, Re David, and Kim G. Larsen. A tutorial on Uppaal. pages 200–236. Springer, 2004.

[2] As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506, November 2004.

The Guided System Development Framework

Jose Quaresma Christian W. Probst Flemming Nielson
Technical University of Denmark
{jncq,probst,nielson}@imm.dtu.dk

1 Introduction

The Service-Oriented Computing paradigm has had significant influence on the Internet, where an increasing number of companies are making their services available. It is now very common to develop systems by specifying the interaction between programs that provide a specific functionality in form of a service, which can be seen as a function that other programs can remotely execute. With the emergence of this paradigm, it is important to provide tools that help system designers to specify the system under development, to enable its easy integration with standard security suites used by industry, to generate code that implements the system's functionality, and to verify the security properties of those systems.

In fact, formal methods provide powerful tools that can be used to verify the security of these systems. However, these tools are not always integrated with the used development environment. That shortcoming is aggravated by the lack of expertise usually necessary to use formal methods and interpret their results. This obstacle can be overcome by a framework that aids and guides the developer on the specification of the system being developed, on choosing the appropriate standard protocols suites that achieve the required security properties, on providing an implementation of the specified system, and also on allowing the verification of its security properties.

In this article, we present the Guided System Development (GSD) framework that provides those functionalities and seamlessly integrates them. The initial idea behind this framework is strongly inspired by CaPiTo [3] and its main contribution: the successful connection of the abstract specification of Service-Oriented Systems with the usage of industry standard suites with the verification of the protocol and generation of code. CaPiTo uses the LySatool [1] to perform the verification of the system and generates system implementations in the programming language C.

2 The Framework

As mentioned above, one of the main ideas behind this framework is to apply CaPiTo's separation of concerns regarding the modelling of communication protocols — especially the separation between the message exchange view and the usage of standard communication suites used in industry — in a simple and intuitive way. This separation of concerns is an essential part of the GSD framework, which enables the specification, implementation and verification of Service-Oriented Systems by having different levels of abstraction and using a functional language similar to the Alice and Bob Notation [5].

As illustrated in Figure 1, the framework can be divided in two phases, the Specification Phase and the Realisation Phase. The former is composed by the Abstract Level and the Standards Level, while the latter is related to the different outputs and translations from the Standards Level.

2.1 The Specification Phase

As mentioned above, the Specification Phase is composed by the two top levels, the Abstract Level and the Standards Level.

In the Abstract Level, the system is described in a language similar to the Alice and Bob notation, with some extensions, such as Receiver Side Actions and Security Modules. Receiver Side Action allow

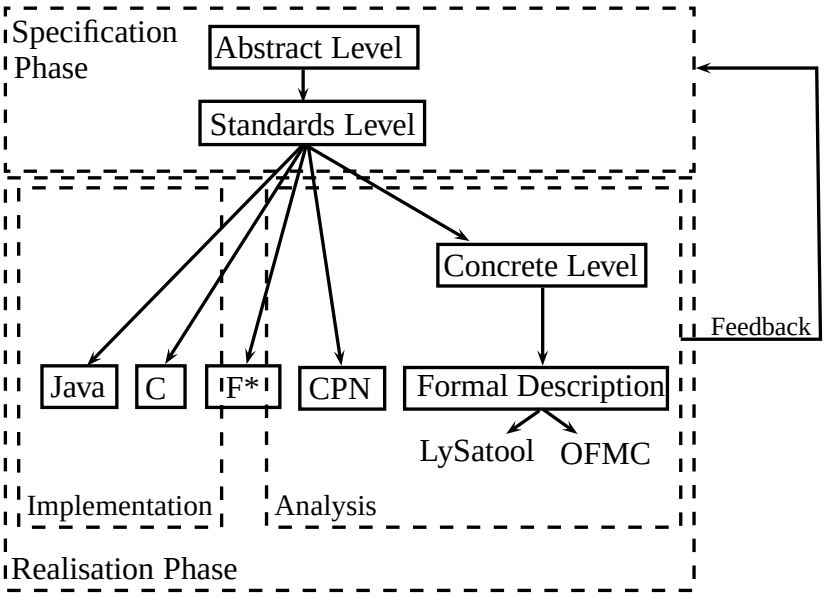


Figure 1: Overview of the Guided System Development framework including some examples of targeted languages and tools

the developer to specify additional actions that will be performed by the receiver of the message, for example, comparison and assignment of values. We believe this is an important extension of standard approaches, because it allows a more complete description of the system. The other main extension is the use of Security Modules, which specify the security requirements of the transmission of some elements of the message exchange. Such requirements are, for example, confidentiality, authenticity, and security. Using these, the developer is able to specify the goals of the system in an early stage of development. These goals, comparable to the ones is BAN logic [2], enable the reasoning about security properties of the described system.

The Abstract Level description of a system is translated into the Standards Level. This translation is performed by identifying Standard Suites that achieve the requirements expressed by the Security Modules. This can be realised by giving suggestions to the developer, depending on the Security Module used in the system description.

2.2 The Realisation Phase

The main goal of the Realisation Phase of the GSD framework is twofold; on one hand, it enables the translation of the system specification into different platforms such as Java, C, or CP-nets. On the other hand, it connects to verification tools, in order to reason about the security properties of the system.

By providing a translation from the system specification into executable code, the developer can easily obtain an implementation of the system, similar to earlier work [7].

Targeting Coloured Petri Nets [4] would add an alternative way of describing the system. CP-nets is a graphical oriented modelling language for the design, specification, simulation and verification of systems. It extends the original Petri nets with the capabilities of the high-level language, allowing the definition of data types — that can be seen as coloured tokens — and the manipulation of data values.

As for the reasoning about the security properties of the system, it can be achieved by translating the system to the Concrete Level — which represents the full message exchange — and verifying its security properties with tools such as LySatool [1] and OFMC [6]. The analyses results allow us to provide some

instant feedback to the system developer regarding the security of the system under development.

Another possibility that we are currently investigating is the translation into F* [8], a dependently typed language for secure distributed programming. This approach would allow to combine executable code with verification of security properties.

2.3 Example

In Figure 2, we show a simple message exchange between Alice and Bob to illustrate the functioning of the framework. We show it for the two levels that constitute the core of the framework and also for the Concrete Level.

While several Standard Suites could be used to implement the goal required in the Abstract Level, we are using TLS, an extremely wide-spread suite, to achieve confidentiality of the message. Confidentiality here means that the receiver (B) is authenticated and that the message can only be seen by the sender (A) and the receiver (B) in the modelled message exchange. The Concrete Level represents the full message exchange. We assume that Alice knows the public key of the certificate authority (CA) prior to the message exchange. Furthermore, both Alice and Bob calculate the resulting symmetric keys sk_{AB} and sk_{BA} (one for each direction of the communication) based on the values of n, m, N. Also, $Finished$ is generated based on these values together with the messages previously exchanged by Alice and Bob. For simplicity reasons, we are not showing the receiver side action and derivation of extra elements.

$$A \rightarrow B : confidential(M)$$

Abstract Level

$$A \rightarrow B : TLS(M)$$

Standards Level

$$\begin{aligned}
A &\rightarrow B : fresh(n) \\
B &\rightarrow A : fresh(m), \{B, pk_B^+\} : pk_{CA}^- \\
A &\rightarrow B : \{fresh(N)\} : pk_B^+, \ \{Finished\} : sk_{AB} \\
B &\rightarrow A : \{Finished\} : sk_{BA} \\
A &\rightarrow B : \{M\} : sk_{AB}
\end{aligned}$$

Concrete Level

Figure 2: Example of a system with a unique message that sends message M in a confidential way

3 Conclusion

With this framework, we provide the developer with an environment that allows a high-level specification of a system (and its goals), the use of Standard Suites to achieve those goals, and the interconnection to different runtime environments and formal approaches. The latter support verification of security properties, the result of which can be used to improve the system specification.

The next step in our research will be to define and implement the core of the framework and then investigate which of the alternatives are more suitable regarding the Concrete Level of the framework. We are currently defining the semantics of the language in the Abstract Level and the translation from that level to the Standards Level.

References

[1] Mikael Buchholtz. *User's Guide for the LySatool version 2.01*. DTU, April 2005.

[2] Michael Burrows, Martin Abadi, and Roger Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8:18–36, February 1990.

[3] H. Gao, F. Nielson, and H.R. Nielson. Protocol Stacks for Services. In *Foundations of computer security*, 2009.

[4] Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.

[5] S. Modersheim. Algebraic Properties in Alice and Bob Notation. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 433 –440, 2009.

[6] Sebastian Mödersheim and Luca Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In *Foundations of Security Analysis and Design V*, volume 5705 of *Lecture Notes in Computer Science*, pages 166–194. Springer Berlin / Heidelberg, 2009.

[7] Jose Quaresma and Christian W. Probst. Protocol implementation generator. *Nordic Conference in Secure IT Systems (NordSec 2010)*, 2010.

[8] Nikhil Swamy, Juan Chen, Cedric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. *The 16th ACM SIGPLAN International Conference on Functional Programming (ICFP 2011), to appear*, September 2011.

Formalising Metamodel Evolution based on Category Theory

Florian Mantz,* Yngve Lamo
Bergen University College
{fma,yla}@hib.no

Alessandro Rossini, Uwe Wolter
University of Bergen
{rossini,wolter}@ii.uib.no

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

Model-driven engineering (MDE) is a branch of software engineering which aims at improving productivity, quality, and cost-effectiveness of software development by shifting the paradigm from code-centric to model-centric activities. MDE promotes models and modelling languages as the main artefacts of the development process and model transformation as the primary technique to generate (parts of) software systems out of models. Models enable developers to reason at a higher level of abstraction, while model transformation restrains developers from repetitive and error-prone tasks such as coding. Although techniques and tools for MDE have advanced considerably during the last decade, several concepts and standards in MDE are still defined semi-formally, which may not guarantee the degree of precision required by MDE.

Models can be specified using general-purpose languages like the Unified Modeling Language (UML) [8], but to fully unfold the potential of MDE, models are often specified using domain-specific languages (DSLs) which are tailored to a specific domain of concern. One way to define DSLs in MDE is by specifying metamodels, which are models that describe the concepts and define the syntax of a DSL. A model is said to *conform to* a metamodel if each element in the model is typed by an element in the metamodel and, in addition, satisfies all constraints of the metamodel.

Models and metamodels undergo complex evolutions during their life cycles. As a consequence, when a metamodel is modified, models conforming to this metamodel need to be migrated in such a way that they conform to the modified version (see Fig. 1). In the literature, this problem is referred to as metamodel evolution [7] or model co-evolution [4].

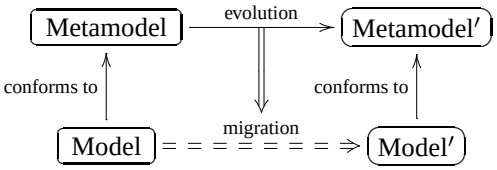


Figure 1: Model co-evolution: Metamodel evolution and model migration

To address this problem, a few prototype tools have been developed that support metamodel evolution in different scenarios, e.g., [7, 9]. However, a uniform formalisation of metamodel evolution is still lacking. The relation between metamodel- and model changes should be formalised in order to allow reasoning about the correctness of migration definitions. In addition, constraints in models and metamodels should also be handled during migration. This work proposes a formal approach to metamodel evolution which addresses some of these challenges. The approach is based on the Diagram Predicate Framework (DPF) [2], a formal diagrammatic specification framework founded on category theory [1] and graph transformation [5]. DPF provides the means to specify models with diagrammatic constraints and defines a conformance relation between models and metamodels which takes into account these constraints [10].

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

In DPF, a model is represented by a *specification* $\mathfrak{S}$. A specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ consists of an *underlying graph* S together with a set of *atomic constraints* $C^{\mathfrak{S}}$ which are specified by means of a *signature* Σ. A signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$ consists of a set of *predicates* $\pi \in \Pi^{\Sigma}$, each having an arity (or shape graph) $\alpha^{\Sigma}(\pi)$, a semantic interpretation, and a proposed visualisation. An atomic constraint (π, δ) consists of a predicate $\pi \in \Pi^{\Sigma}$ together with a graph homomorphism $\delta : \alpha^{\Sigma}(\pi) \to S$ from the arity of the predicate to the underlying graph of the specification.

The semantics of nodes and arrows of a specification has to be chosen in a way which is appropriate for the corresponding modelling environment [11]. In object-oriented structural modelling, it is appropriate to interpret nodes as sets and arrows $\mathsf{X} \xrightarrow{\mathsf{f}} \mathsf{Y}$ as multi-valued functions $f : X \to \wp(Y)$. The semantics of a specification is defined in the fibred way [12]; i.e., the semantics of a specification $\mathfrak{S} = (S, C^{\mathfrak{S}} : \Sigma)$ is given by the set of its instances (I, ι). An instance (I, ι) of a specification $\mathfrak{S}$ consists of a graph I together with a graph homomorphism $\iota : I \to S$ which satisfies the set of atomic constraints $C^{\mathfrak{S}}$.

Metamodel- and model changes can be formalised in DPF as specification transformation rules, which can be regarded as an extension of graph transformation rules [5]. In this work, possible metamodel changes are restricted to a specific set of metamodel evolution/migration rules. The migration rules are derived from metamodel evolution rules by retyping them on the model level; i.e., an isomorphic migration rule is derived from a metamodel evolution rule by replacing each metamodel element by its instance element. This rule is matched as often as possible on a model. This approach can be considered as a special kind of amalgamated graph transformation rule [3] with an empty kernel rule.

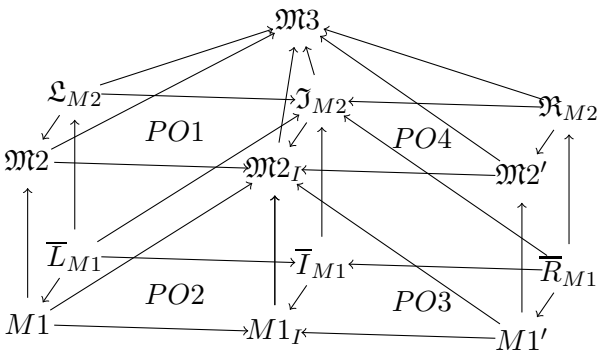


Figure 2: Relations of metamodel- and model changes

Figure 2 shows the graph homomorphisms between a metamodel evolution rule and a model migration rule. These rules are formulated using the cospan double pushout (Cospan DPO) approach [6], which first extends a graph and then reduces it. Equivalence to the original DPO approach is shown by Ehrig et al. in [6]. This approach has been chosen since it allows the models to be adapted in-place. Firstly, a metamodel is extended by the pushout over the span $\mathfrak{M}2 \leftarrow \mathfrak{L}_{M2} \rightarrow \mathfrak{I}_{M2}$ (PO1). Afterwards, a conforming model is extended by PO2 over the span $M1 \leftarrow \overline{L}_{M1} \rightarrow \overline{I}_{M1}$ and then reduced by PO3 over the span $\overline{I}_{M1} \leftarrow \overline{R}_{M1} \rightarrow M1'$. Finally, the metamodel is reduced by PO4 over the span $\mathfrak{I}_{M2} \leftarrow \mathfrak{R}_{M2} \rightarrow \mathfrak{M}2'$. The application sequence of these pushouts allow that models stay type conform during the entire migration process.

Figure 3 shows the graph homomorphisms between a metamodel evolution rule and a model migration rule in more detail. The metamodel evolution rule is represented by the cospan $\mathfrak{L}_{M2} \rightarrow \mathfrak{I}_{M2} \leftarrow \mathfrak{R}_{M2}$, whereas copies of the derived isomorphic migration rule are represented by the family of cospans $L_i \rightarrow I_i \leftarrow R_i$ with $1 \leq i \leq n$. The applicable model migration rule is represented by the cospan $\overline{L}_{M1} \rightarrow \overline{I}_{M1} \leftarrow \overline{R}_{M1}$, which is constructed by the disjoint union. The disjoint union can be charac-

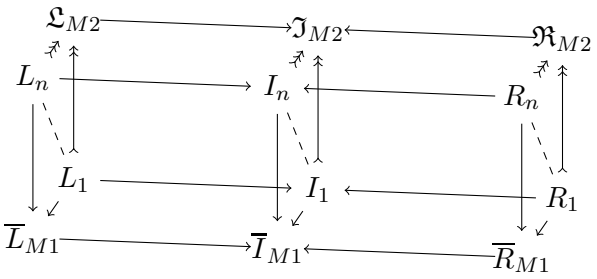


Figure 3: Amalgamated migration rule

terised as coproduct in a corresponding rule category. That fact that the disjoint union is also typed over the metamodel evolution rule can be shown by the universal property of coproducts.

This migration rule deduction strategy is only useful for a subset of metamodel evolution rules. Therefore, the metamodel evolution rules are extended by DPF's atomic constraints. These atomic constraints restrict the possible application of metamodel evolution rules to those cases where the deduction strategy is sufficient. For example, a multiplicity constraint [1..*] added to an arrow of the LHS graph of the metamodel evolution rule prevents this arrow from matching with a metamodel arrow having multiplicity constraint [0..*]. Currently, each arrow $\mathsf{X} \xrightarrow{\mathsf{f}} \mathsf{Y}$ in the metamodel evolution rule is required to be total and surjective, i.e., $|f(x)| \geq 1$ and $\forall y \in Y,\ \exists x \in X : y \in f(x)$. Furthermore, LHS and RHS graphs of metamodel evolution rules with loops and nodes being targets of more than one arrow are not considered for the automatic deduction of migration rules yet.

References

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

[2] Bergen University College and University of Bergen. *Diagram Predicate Framework Web Site*. http://dpf.hib.no/.

[3] E. Biermann, C. Ermel, and G. Taentzer. Formal foundation of consistent EMF model transformations by algebraic graph transformation. *SoSyM (Online First)*, pages 1–24, 2011.

[4] 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.

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

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

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

[8] Object Management Group. *Unified Modeling Language Specification*, May 2010. http://www.omg.org/spec/UML/2.3/.

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

[10] A. Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD thesis, Department of Informatics, University of Bergen, Norway, 2010.

[11] A. Rutle, A. Rossini, Y. Lamo, and U. Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages. In *TOOLS 2009*, volume 33 of *LNBIP*, pages 37–56. Springer, 2009.

[12] U. Wolter and Z. Diskin. From Indexed to Fibred Semantics – The Generalized Sketch File. Technical Report 361, Department of Informatics, University of Bergen, Norway, October 2007.

Parametric WCET Analysis

Björn Lisper

Mälardalen University, Sweden

Abstract

The purpose of Worst-Case Execution Time (WCET) analysis is to compute a safe upper bound to the execution time of a sequential program executing uninterrupted on some given hardware. Such bounds are important when verifying the timing requirements on hard real-time systems. WCET analysis has been an active research topic for the last 20 years, and today there exists a large body of theory, methods, and algorithms. Both academic and commercial tools have emerged during the last decade, and the technique is becoming established in industrial use. Traditional WCET analysis computes a single number. For programs whose execution time varies strongly with the inputs, a single upper bound may provide very large overestimations in most situations since it has to take the program executions for all possible input values into account. It may then be advantageous to have a parametric WCET analysis, which computes the WCET bound as a symbolic formula in the unknown inputs rather than as a single number. When the formula is instantiated for the specific inputs at hand, the resulting number is likely to provide a much tighter bound for the actual WCET. Thus, it is highly interesting to develop good methods and tools for parametric WCET analysis. In this talk we will first give a short primer to WCET analysis. We then give an account for the past, present, and planned future research at Mlardalen University regarding parametric WCET analysis.

Estimating 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.

In this setting, a program execution may exceed the upper bound on the number of transactions the system can afford. Transactions contribute to the resource consumption which 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. In other words, 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.

In this work, we will statically predict resource consumption in connection with transactions by identifying the maximum number of logs produced at any given point in time during the parallel execution of transactions. From that maximum, we can infer information about resource consumption such as memory usage.

* 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

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 guarantee safe commits and estimate the upper bound of resource consumption during its execution. This helps to predict the usage of resources in concurrent transaction systems.
- 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.

Value sensitivity in information flow analysis

Bart van Delft, Chalmers University of Technology, vandeba@chalmers.se
Abstract for NWPT 2011

1 Introduction

Much research in the area of computer security has been focused on regulating access to information (e.g. file permissions, encryption standards etc.), however in this abstract we look at how this information, once obtained, is propagated through a program. In *information flow analysis* it is aimed to detect if sensitive information handled in a program can leak to observers with no access to that information.

In line with other publications on information flow security we use a simple lattice to illustrate the theory, consisting of a HIGH and a LOW security level. We assume that `h`-program variables contain, at the start of the program execution, information of a HIGH security level, whereas `l`-program variables contain information of a LOW security level (which can be observed by an outside user/attacker of the program). During the analysis of a program `p` we want to detect the undesired information flow from `h` to `l`. A program in which such a flow does not occur is said to satisfy the *non-interference* property.

Example 1. We label a program *secure* or *insecure* depending on the satisfaction of the non-interference property:

(i) `l = h` is *insecure* because it leaks information directly from `h` to `l`.

(ii) `if (h > 0) {l = 10} else {l = 20}` is *insecure* because partial information on the value of `h` can be deduced from the value of `l`.

(iii) `if (h > 0) {l = 10} else {l = 20}; l = 0` is *secure* because the final value of `l` does not depend on `h`.

(iv) `if (h > 0) {l = 2} else {l = 2}` is *secure* because the final value of `l` is the same whether the first or second branch has been taken.

(v) `h = 0; l = h` is *secure* because the value of `l` is always 0.

(vi) `l = h; l = l - h` is *secure* because the value of `l` is always 0.

(vii) `if (h > 0) {h = l; l = h}` is *secure* because the value of `l` is not changed.

(viii) `if (h - h + l > 0) {l = 0} else {l = 2}` is *secure* because the value of `h` has no influence on the condition, and thus the final value of `l` is only affected by the value of `l` before execution of the program.

A common approach is to use a type system that adds an additional type to program variables indicating its security level. Via type interference or type checking non-interference is ensured. These systems are always *sound*, i.e. they never classify an insecure program as secure. However to preserve a reasonable degree of automation, compromises need be made with respect to the completeness of these systems, i.e. they may classify secure programs as insecure, or 'unknown'.

Analyses that have trouble correctly labeling secure programs of the kind such as (iii) and (iv) in Example 1, lack *control-flow sensitivity*. Programs of the kind as (v),(vi),(vii) and (viii) require an analysis to have *value sensitivity* in order to correctly label them as secure. In (Bubel et al., 2009)

a different approach is taken by the use of *dependencies*, giving them a benefit over type systems in proving programs such as (iii) and (iv) secure. Some of the programs requiring a value-sensitive analysis are however still labeled as insecure.

Here we show the dependency-based approach and identify how it can be altered such that we achieve a more value sensitive analysis.

2 Sequent calculus

We incorporate the *sequent calculus* (Gentzen, 1934) as used by KeY (Beckert et al., 2007) to perform information flow analysis. A *sequent* is of the form $\varphi_1,\ldots,\varphi_n \Longrightarrow \varphi_m,\ldots\varphi_k$ where φ_i is a first-order dynamic logic formula. A sequent can be abbreviated to $\Gamma \Longrightarrow \Delta$ or to $\Gamma,\varphi_i \Longrightarrow \varphi_j,\Delta$ to single out certain formulas. A sequent can be considered as a meta-formula with implication as its main connective, that is, $Eval(\Gamma \Longrightarrow \Delta) = Eval(\bigwedge\Gamma -> \bigvee\Delta)$. The calculus defines rules that can be used to construct a proof tree. The sequent to prove forms the root of the tree and calculus rules can be applied until all the leaves of the tree are obviously valid. Some example calculus rules are:

$$\textsc{closeTrue}\quad \frac{\mathit{closed}}{\Gamma \Longrightarrow \mathit{true},\Delta} \qquad \textsc{andRight}\quad \frac{\Gamma \Longrightarrow \phi,\Delta \quad \Gamma \Longrightarrow \psi,\Delta}{\Gamma \Longrightarrow \phi\&\psi,\Delta}$$

Since the formulas in the sequents are dynamic logic formulas, they may contain the modality $[\mathtt{p}]\varphi$ denoting that *if* program `p` terminates, φ holds (we do not concern ourselves with non-termination). The term *symbolic execution* is used for calculus rules that move program statements from the box modality to an *update* that describes the state change. For example:

$$\textsc{assignment}\quad \frac{\Gamma \Longrightarrow \{\mathbf{U}\}\{\mathtt{x} := t\}[\ldots]\varphi,\Delta}{\Gamma \Longrightarrow \{\mathbf{U}\}[\mathtt{x = t;}\ldots]\varphi,\Delta}$$

The update $\{\mathbf{U}\}$ is an abbreviation of the preceding updates. An update in front of a formula means that program variables in that formula should be evaluated to the value described by the update. We prefer updates over programs since they have a deterministic behaviour[1].

3 Tracking dependencies

In the programs from Example 1 we see that interference from `h` to `l` indicates that the value of `l` after execution of the program *depends* on the value of `h` at the start of the program. We therefore call the set of variables that influence the final value of `l` the *dependencies* of `l`. Using the strict definition of non-interference, a program can be labeled secure if the dependencies of `l` form a subset of all the variables with security level LOW (in our example setting), i.e. $\Longrightarrow [\mathtt{p}](\mathit{deps}(\mathtt{l}) \subseteq \textsc{Low})$.

Bubel et al. incorporate the sequent calculus to track dependencies as follows. For each program variable `x` an additional variable $\mathtt{x}^{\mathtt{dep}}$ is created that tracks the set of dependencies for `x`. The ASSIGNMENT rule from above is for example translated to:

$$\textsc{assignment}^{\mathtt{dep}}\quad \frac{\Gamma \Longrightarrow \{\mathbf{U}\}\{\mathtt{x} := t \,\|\, \mathtt{x}^{\mathtt{dep}} := \mathit{deps}(t)\}[\ldots]\varphi,\Delta}{\Gamma \Longrightarrow \{\mathbf{U}\}[\mathtt{x = t;}\ldots]\varphi,\Delta}$$

Where *deps* is a function that returns a join of the dependencies of all variables syntactically occuring in t. The assignment `x = y + w` would, in a state where `y` and `w` only depend on themselves, for example result in the update $\mathtt{x}^{\mathtt{dep}} := \mathit{deps}(\mathtt{y}) \cup \mathit{deps}(\mathtt{w}) = \{\mathtt{y}\} \cup \{\mathtt{w}\} = \{\mathtt{y},\mathtt{w}\}$. The supported language

[1] A last-one-wins (or: right-most-one wins) semantics is incorporated to ensure this in the presence of conflicting updates.

also incorporates conditional statements and while loops, and the analysis correctly labels the programs from Example 1, except for (vi) and (viii). The reason for labeling (vi) as insecure can easily be seen when the program is simplified to `l = h - h`. This results in the update $\mathtt{l}^{\mathtt{dep}} := \{\mathtt{h}\}$ since the analysis only considers the *syntactic* term to which `l` is updated.

4 Value sensitivity by equivalence classes

The value sensitivity of the analysis by dependencies as described above can be improved as follows. Instead of making $\mathtt{x}^{\mathtt{dep}}$ directly track the dependencies of `x`, it tracks a Herbrand universe-like version of the term t to which `x` is updated (which we call a *term expression*). E.g. the program `y = x - x` would by symbolic execution result in the update $\mathtt{y}^{\mathtt{dep}} := \mathtt{x}_c -_c \mathtt{x}_c$. To check the dependencies of `y` after symbolic execution of the program, we use a function *unwrap* that takes a Herbrand copy ($\mathtt{x}_c -_c \mathtt{x}_c$) and returns the regular term ($\mathtt{x} - \mathtt{x}$). If we take from this term the variables it syntactically contains ($\{\mathtt{x}\}$) we have the same analytical power as before.

The value-sensitivity that we gain by tracking the term copy works as follows. We define the *equivalence class* of term expressions (the copied terms) to be the same for all term expressions that, when unwrapped, evaluate to the same value independent of the state of the program. That means that for example $\mathtt{x}_c -_c \mathtt{x}_c$ and 0_c belong to the same equivalence class. We denote the equivalence class of a term expression by placing brackets around them, i.e. $[\mathtt{x}_c -_c \mathtt{x}_c] = [0_c]$. Furthermore, we define the evaluation of term expressions as their equivalence class, thus we have that $\mathit{Eval}(\mathtt{x}_c -_c \mathtt{x}_c) = \mathit{Eval}(0_c)$.

With these definitions in place, we can now allow *rewrite rules* between term expressions of the same class, since they evaluate the same. Some example rewrite rules include (where $t_{\mathtt{Exp}}$ is an arbirtray term expression):

$$
\begin{aligned}
t_{\mathtt{Exp}} -_c t_{\mathtt{Exp}} &\rightsquigarrow 0_c \\
1_c *_c t_{\mathtt{Exp}} &\rightsquigarrow t_{\mathtt{Exp}} \\
0_c *_c t_{\mathtt{Exp}} &\rightsquigarrow 0_c \\
t_{\mathtt{Exp}} *_c (v_{\mathtt{Exp}} /_c w_{\mathtt{Exp}}) &\rightsquigarrow (t_{\mathtt{Exp}} *_c v_{\mathtt{Exp}}) /_c w_{\mathtt{Exp}}
\end{aligned}
$$

These rewrite rules could also be allowed in the opposite direction, but this would for most rules not be beneficial for a value sensitive analysis. We can now rewrite the tracked $\mathtt{y}^{\mathtt{dep}} := \mathtt{x}_c -_c \mathtt{x}_c \rightsquigarrow 0_c$ and conclude that the set of variables in the unwrapped term expression is $\{\}$, thus that `y` does not depend on any variable.

Incorporating a similar mechanism for conditional statements and updating the calculus rules accordingly we can now show that our analysis correctly classifies all the programs from Example 1. In future research we plan to demonstrate an implemented version of this analysis in the KeY System.

References

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract Interpretation of Symbolic Execution with Explicit State Updates. In *Revised Lectures, 7th International Symposium on Formal Methods for Components and Objects (FMCO 2008)*, volume 5751 of *LNCS*, pages 247–277. Springer, 2009.

Gerhard Gentzen. Untersuchungen über das logische Schließen I. *Mathematische Zeitschrift*, (39):176–210, 1934.

Static Analysis of Bounded Polyhedra

Stefan Bygde
Mälardalen University
Västerås, Sweden
stefan.bygde@mdh.se

Björn Lisper
Mälardalen University
Västerås, Sweden
bjorn.lisper@mdh.se

Niklas Holsti
Tidorum Ltd
Hesinki, Finland
niklas.holsti@tidorum.fi

Abstract

We present a method for polyhedral abstract interpretation which derives fully bounded polyhedra for every step in the analysis. Contrary to classical polyhedral analysis, this method is sound for integer-valued variables stored as fixed-size binary strings; wrap-arounds are correctly modelled. Our work is based on earlier work by Axel Simon and Andy King but aims to significantly reduce the precision loss introduced in their method.

1 Introduction

A commonly used application of program analysis is to derive which numerical values the program variables can take at each point in the program. This is typically done using abstract interpretation [2] and some numerical abstract domain to get an approximation of the possible values. Abstract interpretation derives a superset of the possible values that each variable can take at each point in the program. Since exact information about the semantics of a program is uncomputable, abstract interpretation introduces a sound approximation of the possible values. The nature of the abstraction is implemented via *abstract domains* which determine which properties of the values should be abstracted. An abstract domain consists of an abstract representation of a set of program states (called abstract environments) and defines transfer functions for each construct in the programming language. The transfer functions describe how a construct changes the abstract environments. Abstract interpretation is then done by iterating and propagating the abstract environments through the program via the transfer functions until no changes occur (fixed-point iteration).

Most numerical domains suggested in the literature are based on the unsound assumption that variables can take arbitrary integer values. In real applications, a value is usually stored as a fixed-sized binary string. Analyses based on an abstract domain that assumes unbounded integers fail to model effects such as wrap-arounds. One of the most common abstract domains is the domain of *convex polyhedra* [3], which is a powerful domain that captures linear relationships between the values of the variables. The set of possible environments [1] at a program point is modelled as the set of solutions to a system of linear inequalities in $\mathbb{R}^n$ space. Each dimension corresponds to one integer-valued variable. Each (integer) solution corresponds to a possible environment at one point in a program. The set of solutions forms a convex polyhedron in $\mathbb{R}^n$ space, hence the name. Since a linear inequality may contain several variables, this domain is able to capture linear relations between variables, which is important to achieve precision.

The polyhedral domain is also based on the assumption that variables can take arbitrary integer values, so it is unsound for programs containing wrap-arounds. Simon and King [5] suggest a modification to the domain which makes it sound for wrapping integers, but this modification introduces a lot of imprecision.

2 Contributions

Our work aims to provide a sound and precise polyhedral analysis for integers with possible wrap-arounds. Our method builds on earlier work by Simon and King [5], but we attempt to minimise the

[1] In this context, an environment is an assignment of an integer value to each integer-valued variable.

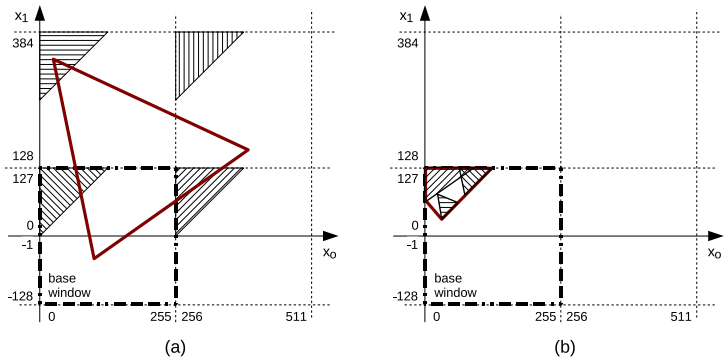


Figure 1: The picture on the left (a), shows a polyhedron before wrapping. The base window is shown outlined by a dot-dashed square. The grid of variously hatched triangles shows the condition $x_0 \leq x_1$ taken as a signed comparison of the 8-bit unsigned residue of x_0 with the 8-bit signed residue of x_1. The polyhedron intersects three components of this condition. To the right (b), the intersections of the condition with the unwrapped polyhedron are shown, shifted to the base window, and their convex hull, which is the resulting wrapped polyhedron.

introduced imprecision of their method. In classical polyhedral analysis, and indeed in Simon and King's approach, it is possible to derive polyhedra which are unbounded in some dimension. Unbounded polyhedra unfortunately introduce a lot of imprecision in Simon and King's approach. Thus, we devise a method which makes sure that every analysis step derives a fully bounded polyhedron. This makes our method potentially more precise than [5] and allows analysis with the polyhedral domain without compromising soundness or causing too much imprecision. The details of our method are available in [1].

3 Wrapping Polyhedra

The method in [5] interprets the solutions to the system of linear inequalities (i.e., the points inside a polyhedron) modulo 2^N, where N is the number of bits in a variable, with proper adjustment for signed variables. Performing abstract interpretation while using this interpretation works well since linear assignments commute with modular residue. However, linear inequalities do not commute, and therefore an explicit wrapping of the polyhedron is required whenever a linear comparison is made.

The first step of wrapping is to partition $\mathbb{R}^N$ into "windows" as in Figure 1. The linear inequality is repeated in each window to capture the comparison between an integer which might have been wrapped (i.e., lies outside the "base window"). The wrapping is then done by intersecting the partitioned polyhedron parts with the repeated inequalities, shifting the results to the base window, and finally computing the smallest polyhedron that covers all the shifted parts of the polyhedron (see the right part of Figure 1). Unbounded polyhedra are problematic for this wrapping strategy, as they would require intersecting an infinite number of polyhedra. Thus, for any unbounded variable, any relational information is discarded and the variable is simply bounded by the base window size in its dimension. Unfortunately, unbounded polyhedra commonly occur during polyhedral analysis of typical programs.

4 Bounded Polyhedra

Our approach is to use the wrapping strategy to make sound analyses, while preventing unnecessary precision loss due to unbounded polyhedra. Most of the precision loss comes from unbounded polyhedra,

so our approach is to make the polyhedra in the analysis fully bounded. There are three phases of classic polyhedral analysis which can make a polyhedron unbounded: at the initial analysis state, at any non-linear assignment of a variable and at widenings. Widening is an acceleration technique required for most abstract domains in order to terminate the analysis quickly [2].

In classical polyhedral analysis, nothing is assumed to be known about the initial state of the program variables. This is symbolised by an infinite unbounded polyhedron (or equivalently, the system of linear inequalities is empty). To make this state bounded we impose type-bounds on the initial state. Type-bounds are simply the min and max values for any signed or unsigned integer-variable represented by N bits. For example, the type-bounds for an unsigned 8-bit variable x would be: $0 \leq x \leq 255$. Thus, the initial state can safely be bounded.

The polyhedral domain, by nature, has problems with non-linearity. A non-linear assignment of a variable causes the analysis to drop all assumptions about it (i.e., losing all relational information about it). This results in a polyhedron which is unbounded in the dimension the assigned variable represents. However, since relations to other variables in this case are unknown, it is still safe to impose type-bounds for this variable, as we did in the initial state, thus making the polyhedron bounded.

Finally, we have the widening. Widening for polyhedra consists of removing those inequalities, at the widening point, that do not hold in two consecutive iterations of the analysis. Removing inequalities often results in an unbounded polyhedron. Since the polyhedron may still contain relational information, it is not safe to impose type bounds after widening.

Our approach is to do widening in conjunction with wrapping to make sure that the polyhedron remains within all type-bounds. To avoid unnecessary wrapping that could cause imprecision we apply widening only at the points where wrapping must be done - at conditional branches that depend on linear comparisons. This makes it safe to impose the type-bounds on the polyhedron after the widening (this is equivalent to making a limited widening [4] over the type-bounds), thus making the polyhedron fully bounded after widening. This, together with imposing type-bounds for variables with no relational information, results in fully bounded polyhedra in every step of the analysis.

5 Conclusion

We have developed a polyhedral analysis which is safe for wrapping integers while not losing too much precision. The core of the method is to make sure that the analysis derives fully bounded polyhedra in order to avoid unnecessary precision loss when wrapping. An implementation of the method is ongoing and a thorough comparison to Simon and King's original method is planned. Due to the more precise widening, our analysis may take a little longer to stabilise, but potentially with more precision.

References

[1] Stefan Bygde, Björn Lisper, and Niklas Holsti. Fully bounded polyhedral analysis of integers with wrapping. In *International Workshop on Numerical and Symbolic Abstract Domains (NSAD)*, September 2011.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[3] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[4] Nicolas Halbwachs. Delay analysis in synchronous programs. In Costas Courcoubetis, editor, *CAV*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346. Springer, 1993.

[5] Axel Simon and Andy King. Taming the wrapping of integer arithmetic. In *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin / Heidelberg, 2007.

Towards a Real-Time, WCET Analysable JVM Running in 256 kB of Flash Memory

Stephan Korsholm
VIA University College
8700 Horsens, Denmark
sek@viauc.dk

Kasper Søe Luckow
Aalborg University
9220 Aalborg, Denmark
luckow@cs.aau.dk

Bent Thomsen
Aalborg University
9220 Aalborg, Denmark
bt@cs.aau.dk

1 Introduction

The Java programming language has recently received much attention in the real-time systems community as evidenced by the wide variety of initiatives, including the Real-Time Specification for Java[8], and related real-time profiles such as Safety-Critical Java[6], and Predictable Java[3]. All of these focus on establishing a programming model appropriate for real-time systems development. The motivation for the profiles has been to further tightening the semantics, for accommodating static analyses, such as Worst Case Execution Time (WCET) analysis that serves an integral role in proving temporal correctness.

Evidently, the presence of the Java Virtual Machine (JVM) adds to the complexity of performing WCET analysis. To reduce the complexity, a direction of research has focused on implementing the JVM directly in hardware[9]. To further extend the applicability of real-time Java, we want to allow software implementations of the JVM executed on common embedded hardware, such as ARM and AVR, while still allowing the system to be amenable to static analyses. This necessarily demands that the JVM is amenable to static analyses as well, which is difficult, since the JVM specification is rather loosely defined. Specifically, the JVM specification emphasises on *what* a JVM implementation must do whenever executing a Java Bytecode, but leaves *how* unspecified. This makes JVM vendors capable of tailoring their implementation to their application domain.

In our recent research, we have developed a WCET analysis tool called Tool for Execution Time Analysis of Java bytecode (TetaJ)[5], which allows for taking into account a software implemented JVM and the hardware. The development of TetaJ has made us explicitly reason about JVM design accommodating WCET analysis. The contribution of this paper is to present our preliminary research efforts in making Java tractable for real-time embedded systems on more common execution environments by elaborating on *how* a real-time JVM must handle certain issues. Constraining the degree of freedom of the real-time JVM vendors is a necessity to ensure, that the application running on the JVM is temporally correct, since the WCET is often obtained using static analyses relying on predictable behaviour.

2 TetaJ

TetaJ employs a static analysis approach where the program analysis problem of determining the WCET of a program is viewed as a model checking problem. This is done by reconstructing the control flow of the program and the JVM implementation, and generate a Network of Timed Automata (NTA) amenable to model checking using the state-of-the-art UPPAAL model checker[1]. The NTA is structured such that the model checking process effectively simulates an abstract execution of the Java Bytecode program on the particular JVM and hardware.

TetaJ has proven suitable for iterative development since it analyses on method level, and because analysis time and memory consumption are reasonably low. In a case study[5, 2], an application consisting of 429 lines of Java code was analysed in approximately 10 minutes with a maximum memory consumption of 271 MB. The case study is based on the Atmel AVR ATmega2560 processor and the

Hardware near Virtual Machine (HVM)[1] which is a representative example of a JVM targeted at embedded systems.

Currently, TetaJ provides facilities for automatically generating an NTA representing the JVM by the provision of the HVM executable. The METAMOC[4] hardware models are reused in TetaJ thereby having the desirable effect that TetaJ can benefit from the continuous development of METAMOC.

We have strong indications that TetaJ produces safe WCET estimates, that is, estimates that are at least as high as the actual WCET and TetaJ may therefore be appropriate for analysing hard real-time Java programs. As to the precision, we have results showing that TetaJ produces WCET estimates with as low as 0.6% of pessimism[5].

3 Hardware near Virtual Machine

The HVM is a simplistic and portable JVM implementation targeted at embedded systems with as low as 256 kB of flash memory and 8 kB of RAM and is capable of running bare-bone without operating system support. To support embedded systems development, the HVM implements the concept of *hardware objects*[7], that essentially prescribe an object-oriented abstraction of low-level hardware devices, and allow for first-level interrupt handling in Java space.

The HVM employs iterative interpretation for translating the Java Bytecodes to native machine instructions. The interpreter itself is compact, and continuously fetches the next Java Bytecode, analyses it, and finally executes it. The analyse and execute stages are implemented by means of a large switch-statement with cases corresponding to the supported Java Bytecodes.

A special characteristic of the HVM is that the executable is adapted to the particular Java Bytecode program. Specifically, the Java Bytecode of the compiled program is encapsulated in arrays within the HVM itself. This, however, does not affect the behaviour of the interpreter, and is merely a way of bundling the Java Bytecode with the HVM into a single executable.

4 Preliminary Design Criteria for a Predictable HVM

During the development of TetaJ, the implementation of the HVM has been inspected and modified according to the needs of WCET analysis. Some modifications simply comprise bounding the number of loop iterations while others require more elaborate solutions to be developed. In the following, we present our experiences with modifying the HVM towards predictable and WCET analysable behaviour.

4.1 Eliminating Recursive Solutions

Some Java Bytecode implementations are intuitively based on recursive solutions. Specifically, the Java Bytecodes responsible for method invocations such as *invokevirtual* employ a recursive approach.

The heart of the HVM is the *methodInterpreter* which implements the interpretation facilities. Whenever e.g. *invokevirtual* is executed, the *methodInterpreter* is recursively called to process the code of the invoked method. This, however, is undesirable seen from a static WCET analysis perspective, since it is difficult to statically determine the depth of the recursive call. The problem is circumvented by introducing an iterative approach and introduce the notion of a call stack and stack frames. Using this solution, a stack frame containing the current call context, that is, stack pointer, program counter etc. are pushed onto the stack, and the *methodInterpreter* simply continues iteratively fetching Java Bytecodes from the called method. When the particular method returns, the stack is popped and the context restored to the point prior to the method invocation.

[1] http://www.icelab.dk

4.2 Reducing Pessimism of the Class Hierarchy

Since Java is strongly typed, type casts produce the *checkcast* Java Bytecode which is responsible for iteratively checking the class hierarchy to determine whether the type cast is type compatible. Another example is the *instanceof* operator which similarly consults the class hierarchy iteratively. Establishing a tight bound that applies for every part of the class hierarchy cannot be done statically. Instead it is only possible to establish a global bound corresponding to the maximum depth of the class hierarchy. This gives rise to pessimism that affects the resulting WCET extensively.

This problem has been resolved by harnessing that the HVM is adapted to the particular application. Because this process is performed prior to runtime, it is possible to exercise how the class hierarchy is built and construct a matrix specifying how classes are interrelated. The matrix will be incorporated in the final executable, and can be used for determining type compatibility among classes in constant time, by simply looking up the matrix.

4.3 Constant Time Analyse Stage

Different compilers and different optimisation levels may or may not implement a sufficiently large switch-statement as a look-up table. Because of this uncertainty, we have replaced the analyse stage in the *methodInterpreter* to ensure that this stage is performed in constant time regardless of compiler and optimisation levels. The replacement consists of extracting the individual Java Bytecode implementations from the switch-statement into respective functions. This also has the desirable side-effect that they are easily located in the disassembled HVM executable. An array of function-pointers to each of these functions substitutes the original switch-statement, thereby allowing for constant access time to each of the Java Bytecode implementations using the opcodes as look-up keys.

References

[1] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - A Tool Suite for Automatic Verification of Real-time Systems. *Hybrid Systems III*, pages 232–243, 1996.

[2] Thomas Bøgholm, Christian Frost, Rene Rydhof Hansen, Casper Svenning Jensen, Kasper Søe Luckow, Anders P. Ravn, Hans Søndergaard, and Bent Thomsen. Harnessing theories for tool support. *Submitted for publication: Innovations in Systems and Software Engineering*, 2011.

[3] Thomas Bøgholm, René R. Hansen, Anders P. Ravn, Bent Thomsen, and Hans Søndergaard. A predictable java profile: Rationale and implementations. In *JTRES '09: Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 150–159, New York, NY, USA, 2009. ACM.

[4] Andreas E. Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In Björn Lisper, editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 113–123. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[5] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen. WCET Analysis of Java Bytecode Featuring Common Execution Environments. *Accepted for publication: The 9th International Workshop on Java Technologies for Real-time and Embedded Systems - JTRES 2011*, 2011.

[6] JSR302. The java community process, 2010. `http://www.jcp.org/en/jsr/detail?id=302`.

[7] Stephan Korsholm, Anders P. Ravn, Christian Thalinger, and Martin Schoeberl. Hardware objects for java. In *In Proceedings of the 11th IEEE International Symposium on Object/component/serviceoriented Real-time distributed Computing (ISORC 2008*. IEEE Computer Society, 2008.

[8] Oracle. RTSJ 1.1 Alpha 6, release notes, 2009. `http://www.jcp.org/en/jsr/detail?id=282`.

[9] Martin Schoeberl. JOP: A Java optimized processor. In *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *LNCS*, pages 346–359, Catania, Italy, November 2003. Springer.

An Optimal Resource Sharing Protocol for Generalized Multiframe Tasks

— Extended abstract —

Pontus Ekberg, Nan Guan, Martin Stigge, Wang Yi
Uppsala University, Sweden
Email: {pontus.ekberg | nan.guan | martin.stigge | yi}@it.uu.se

***Abstract*—We consider sharing of non-preemptable resources in real-time task models with flexible job release patterns. Resource sharing is an inherent property of many real-time systems. At the same time, flexible task models are needed to precisely express their workloads. Exact analysis and optimal scheduling of systems with shared resources have been available only for relatively simple task models, such as the sporadic task model.**

We propose a new algorithm for scheduling systems with shared resources. The key idea behind the algorithm is to take the tasks' structures into account when predicting possible resource contention. We show that the algorithm is optimal for scheduling *generalized multiframe* task sets with shared resources. We also present an efficient feasibility test for such systems, and show that the test is both sufficient and necessary.

I. Introduction

Processes in real-time systems often compete for *shared resources*, such as peripheral devices or global data structures that must be accessed in a mutually exclusive manner. To avoid deadlocks and low processor utilization, we need scheduling algorithms that handle the resource sharing.

Well-established solutions to the resource sharing problem exist in the context of *sporadic* task sets. The popular instantiation of the stack resource policy [1], often called EDF+SRP [2], is even optimal for such workloads. Unfortunately, EDF+SRP is not optimal for more flexible task models, such as the *generalized multiframe* (GMF) task model [3].

The flexible structure of GMF tasks, in combination with shared resources, is the main source of difficulty in finding an optimal scheduling strategy. To make optimal scheduling decisions at run-time, we must be aware of the tasks' structures and predict which behaviors each task may display in the near future.

The goal of this work is to show how to analyze and schedule GMF task sets with shared resources. We introduce an efficient technique, which takes the tasks' structures into account, to predict possible resource contention at run-time and thereby determine the urgency of unlocking currently used resources. Based on this technique we propose a new scheduling algorithm and show that it is well suited for scheduling such workloads. The main contributions include:

- We propose the *virtual deadline protocol* (VDP) for handling shared resources, and combine it with *earliest deadline first* (EDF) to form the EDF+VDP scheduling algorithm. We prove that EDF+VDP has the following properties:
  - It is *optimal* for scheduling GMF task sets with shared resources, in the sense that it successfully schedules all feasible task sets.
  - It is deadlock-free, and it enables efficient implementations because there is at most one preemption per job release and all jobs in the system can share a common run-time stack.
- We derive a sufficient and necessary feasibility test for GMF task sets with shared resources. This test is in the same complexity class as the known feasibility tests for sporadic tasks [4] and GMF task sets *without* resources [3], i.e., pseudo-polynomial for bounded-utilization task sets.

II. Preliminaries

A. The Generalized Multiframe Task Model

The GMF task model [3] is a generalization of the well-known *sporadic* [4] task model. Like a sporadic task, a GMF task releases a sequence of *jobs*. However, the jobs released by a GMF task do not all need to have the same parameters (e.g., execution time and deadline). Instead, a GMF task cycles through a sequence of *job types*, which specify the parameters of the jobs that are released.

A natural way of representing a GMF task is to use a directed cycle graph, where the vertices represent the job types, and the arcs specify the order in which jobs are released (as well as the minimum delay between consecutive job releases). Formally, a GMF task set τ is defined as follows:

- Each task $T \in \tau$ is a *directed cycle graph*, with vertices $V(T)$ and arcs $A(T)$.
- Each vertex $v \in V(T)$ is called a *job type* and is labeled with a pair $\langle E(v), D(v) \rangle$. For each *job* that is of type v, $E(v) \in \mathbb{N}_{>0}$ is an upper bound on its required execution time, and $D(v) \in \mathbb{N}_{>0}$ is its relative deadline.
- Each arc $(u, v) \in A(T)$ is labeled with a minimum inter-release separation time $P(u, v) \in \mathbb{N}_{\geqslant 0}$.
- One vertex $v_0 \in V(T)$, denoted $S(T)$, is called the *start vertex* of T.

When the system is running, each task T releases a possibly infinite sequence of jobs $[J_0, J_1, J_2, \ldots]$, where each job corresponds to one of T's job types. Intuitively, a job sequence is generated by "walking" through the graph of T, starting at vertex $S(T)$. Every time a vertex is visited, a job of the corresponding job type is released. Before the next vertex can be visited, the task must wait for at least the minimum inter-release separation time labeled on the arc leading there.

Formally, each job J_i in a job sequence is specified by a triple $(r(J_i), e(J_i), d(J_i)) \in \mathbb{R}^3$, where $r(J_i)$ is the job's absolute release time, $d(J_i)$ its absolute deadline and $e(J_i)$ its execution time requirement. A job sequence is said to be *generated* by T if and only if there is a path $[v_0, v_1, v_2, \ldots]$ through T such that the following hold for all $i \geqslant 0$:

1) $v_0 = S(T)$,
2) $r(J_{i+1}) \geqslant r(J_i) + P(v_i, v_{i+1})$,
3) $e(J_i) \leqslant E(v_i)$,
4) $d(J_i) = r(J_i) + D(v_i)$.

A job sequence is generated by a task set τ if and only if it is an interleaving of job sequences generated by the tasks $T \in \tau$.

We assume that the tasks satisfy the l-MAD property [3]: $D(u) \leqslant P(u, v) + D(v)$. This property guarantees that all jobs released by the same task have their (absolute) deadlines ordered in the same order as their release times.

B. Modeling Shared Resources

In the plain GMF model described above, all jobs are completely independent; there is no way to model contention between jobs for shared resources. We extend the GMF model to include non-preemptable shared resources, which allows us to express which resources may be used by jobs of each job type, and for how long. We refer to the extended model as the GMF-R task model.

When a job is granted access to a resource, we say that it *locks* the resource, and then *holds* it for some time before finally *unlocking* it. If a resource is already held by some job, it cannot be locked again until it has been unlocked by the job holding it. Note that a job may be preempted while holding a resource, but no other job may use that resource until it is unlocked.

Each job type has a *worst-case access duration* to each resource. After a resource is locked, the job will execute for at most this duration before unlocking it again. We do not assume any a priori knowledge about exactly *when* a job locks a resource.

Formally, a GMF-R task set is a triple (τ, ρ, α), such that

- τ is a GMF task set,
- ρ is a set of resources,
- $\alpha : V(\tau) \times \rho \to \mathbb{N}_{\geqslant 0} \cup \{\bot\}$ is a function mapping job types and resources to their worst-case access durations,

where $V(\tau) = \bigcup_{T \in \tau} V(T)$ is the set of *all* job types in τ.

The worst-case access duration of jobs of type v to resource $R \in \rho$ is given by $\alpha(v, R)$. If $\alpha(v, R) = \bot$, then jobs of type v do not use resource R.[1] Otherwise, $\alpha(v, R) \leqslant E(v)$ is assumed. We let $\alpha^{\max}(T, R)$ denote the maximum access duration to resource R by any job type in task T. Fig. 1 shows an example GMF-R task set with two GMF tasks and two shared resources (only $\alpha(v, R) \neq \bot$ are shown).

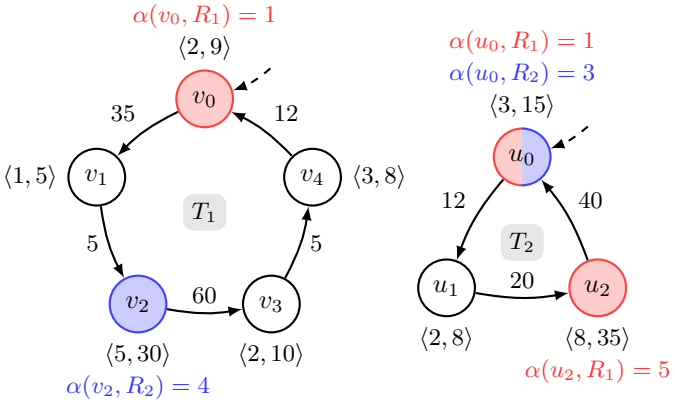


Fig. 1. A GMF-R task set with two tasks and shared resources R_1 and R_2.

A single job may use several different resources, possibly at the same time, but resource accesses must be *properly nested*. That is, if a job locks resource R_1 and afterwards locks R_2 before unlocking R_1, it must unlock R_2 before unlocking R_1. A job may also lock the same resource several times during its execution (each time holding it for at most the worst-case access duration). A job must unlock all resources that it holds before finishing execution.

C. Scheduling Algorithms and Feasibility

For a job sequence to be successfully scheduled, all jobs must finish their execution before their deadlines. That is, each job J in the sequence must be exclusively executed for $e(J)$ time units (not necessarily continuously) between $r(J)$ and $d(J)$. A job is said to be *active* during the time interval between its release time and the time point where it finishes execution.

A *scheduling algorithm* decides at each time point which active, non-blocked job (if any) to execute. A scheduling algorithm can know the current system state and how jobs have been released in the past. It does not know what will happen in the future, other than what is specified by the task model.

Definition II.1 (Feasibility and Optimality). *A* GMF-R *task set* (τ, ρ, α) *is* feasible *if and only if there exists some scheduling algorithm that can successfully schedule all job sequences generated by* τ*, for all legal access patterns to the resources in* ρ *by jobs in the sequence.*

A scheduling algorithm is optimal *if and only if it can successfully schedule all feasible task sets.*

[1] Note that $\alpha(v, R) = 0$ is useful to express that jobs of type v can be forbidden to execute while some other job holds R, but do not hold R themselves. This can be used to model non-preemptable sections in jobs.

III. The Virtual Deadline Protocol

The virtual deadline protocol (VDP) is a resource sharing protocol, designed to extend the earliest deadline first (EDF) scheduling algorithm to handle shared resources. We call the resulting scheduling algorithm EDF+VDP. We will show that EDF+VDP is an optimal scheduling algorithm for GMF-R task sets.

EDF+VDP uses what we call *virtual deadlines* to schedule jobs. It schedules jobs in a similar way to EDF, but uses virtual deadlines instead of absolute deadlines for scheduling decisions. That is, at each time point, EDF+VDP chooses to run the job with the earliest virtual deadline. It is then up to the VDP part of EDF+VDP to assign virtual deadlines to jobs in a way that guarantees the desired properties. It does this by potentially lowering the virtual deadlines (and thereby increasing the priorities) of jobs that are currently holding resources. The virtual deadline of a job therefore represents not only the urgency of the job itself, but also the urgency of releasing the resources that the job is currently holding. To assign virtual deadlines in an optimal way, we must be able to determine how urgent it is that a certain resource becomes unlocked. We capture this urgency by introducing the concept of a *resource deadline*, which is described in the following section.

A. Resource Deadlines

VDP relies on the idea that we can predict, at any time, exactly the earliest future time point where some not-yet-released job *can* have a deadline. In particular, we are interested in the deadlines of future jobs that may need some resource R. The earliest possible such deadline we call the resource deadline of R.

Definition III.1 (Resource deadline)**.** *The resource deadline $\Delta(R,t)$ of resource R at time point t is exactly the earliest time point when some job that is not yet released at t and that may need R can* potentially *have a deadline, without violating the semantics of the task model.*

In other words, if a task set (τ, ρ, α) has released a sequence of jobs $[J, J', \ldots, J'']$ up to time point t, then $\Delta(R,t)$ is the smallest value such that the following is satisfied, for some potential future job J''':

1) J''' may use R,
2) $\Delta(R,t) = d(J''')$,
3) $r(J''') \geqslant t$,
4) some $[J, J', \ldots, J'', \ldots, J''']$ is *generated* by τ.

Note that no future job that uses R actually has to get a deadline at $\Delta(R,t)$, as long as it is *possible*, given the task model and the system state at time t. We will show how resource deadlines can be efficiently computed at run-time.

Example III.2. *Consider the task system in Fig. 1 and the following run-time scenario, illustrated in Fig. 2. At time 15, we want to know the resource deadline $\Delta(R_1, 15)$. The latest job released by task T_1 was of type v_3 at time 11, and the latest job released by task T_2 was of type u_1 at time 2.*

We can see that the next job of T_1 that may need R_1 is of type v_0. The earliest possible deadline of the next job of type v_0 is at $11 + 5 + 12 + 9 = 37$. Similarly, the next job of T_2 that may need R_1 is of type u_2, and can have a deadline earliest at time $2 + 20 + 35 = 57$. The earliest possible time when some future job that needs R_1 may have a deadline is therefore at time 37, and $\Delta(R_1, 15) = 37$.

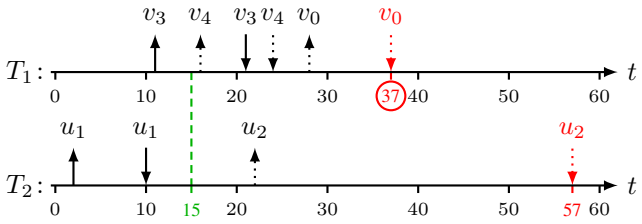


Fig. 2. At time 15 we want to know the resource deadline for R_1. The solid arrows indicate release times and deadlines of the latest jobs from T_1 and T_2 (of types v_3 and u_1, respectively). The dotted arrows indicate the earliest possible release times and deadlines of future jobs.

B. The EDF+VDP Scheduling Algorithm

In EDF+VDP we use virtual deadlines to represent the urgency of executing jobs. The urgency of executing a job depends not only on the job itself (i.e., its absolute deadline), but also on whether the resources that it holds might be needed by some other job. We introduced resource deadlines to capture this latter aspect of the urgency.

By combining these aspects of urgency, we can now present the complete EDF+VDP scheduling algorithm, which is defined by the following four rules:

1) When a job J is released, it gets a virtual deadline $v(J)$ equal to its absolute deadline:
$$v(J) \leftarrow d(J).$$
2) When a job J locks a resource R at time t, $v(J)$ is updated based on the resource deadline $\Delta(R,t)$:
$$v(J) \leftarrow \min(v(J), \Delta(R,t)).$$
3) When a job unlocks a resource, it regains the virtual deadline that it had immediately before locking that resource.
4) At each time point, EDF+VDP executes the active job J with the earliest virtual deadline $v(J)$. If several jobs share this earliest virtual deadline, then those jobs are executed in first-come, first-served order.

References

[1] T. Baker, "A stack-based resource allocation policy for realtime processes," in *RTSS*, 1990, pp. 191–200.
[2] S. Baruah, "Resource sharing in EDF-scheduled systems: A closer look," in *RTSS*, 2006, pp. 379–387.
[3] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Systems*, vol. 17, pp. 5–22, 1999.
[4] A. Mok, "Fundamental design problems of distributed systems for the hard real-time environment," Cambridge, MA, USA, Tech. Rep., 1983.

Adaptive Task Automata: A Framework for Verifying Adaptive Embedded Systems

Leo Hatvani, Paul Pettersson, Cristina Seceleanu
Mälardalen University, 721 23, Västerås, Sweden
{leo.hatvani, paul.pettersson, cristina.seceleanu}@mdh.se

1 Introduction

Adaptive embedded systems are systems that must be capable to dynamically reconfigure in order to adapt to e.g., changes in available resources, user- or application driven mode changes, or modified quality of service requirements. The possibility to adapt provides flexibility that extends the area of operation of embedded systems and potentially reduces resource consumption, but also poses challenges in many aspects of systems development, including system modeling, scheduling, and analysis.

In embedded systems, tasks are usually assumed to execute periodically according to classical real-time scheduling policies such as rate monotonic scheduling, other fixed priorities, earliest deadline first, or first-in first out [3]. For systems with non-periodic tasks or non-deterministic task behaviors fewer general results exists. Automata models have been proposed to relax some of the assumptions on the arrival patterns of tasks. In the model of *task automata* (or timed automata with tasks) [7, 5], the release patterns of tasks are modeled using *timed automata* [1], such that a set of tasks with known parameters is released at the time point an automata location is reached. It has been shown that the corresponding schedulability problem for this bigger class of possible release patterns is decidable, i.e., the problem of checking if for all possible traces of a task automata, the released tasks are schedulable (or not), assuming a given scheduling policy. The theory is implemented in the TIMES tool [2].

In this work, we propose a framework for modeling and analysis of *adaptive real-time embedded systems* based on the model of task automata (see Section 2). Our extension allows for modeling of, e.g., adaptive embedded systems in which decisions to admit further tasks is based on available CPU resources, or systems in which tasks with high quality of service can occasionally be replaced with alternative lower quality tasks when the CPU load is too high. We use a smartphone system example to illustrate our approach (sections 3 and 4).

2 Adaptive Task Automata

Adaptive task automata are an expansion on the framework of finite state automata. First, finite state automata are extended with real-valued clocks to become timed automata [1]. Next, in the work by Fersman et al. [7] the notion of timed automata has been augmented by associating states of the automata with releases of tasks (executable programs) resulting in task automata.

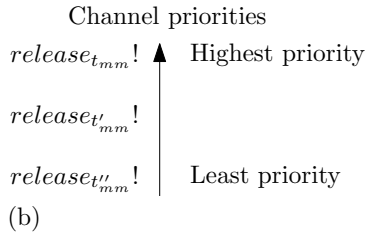
In the same work, Fersman et al. have proved that it is possible to check the schedulability of a model created using task automata, under the assumption that the computation times, hard deadlines and priorities of the tasks are known.

Choice of the next state in a task automata is regulated by the guards on the edges between states. These guards are conjunctions of formulas of type $x_i \sim C$ or $x_i - x_j \sim D$ where $x_i, x_j \in \mathscr{C}$ are real-valued clocks, $\sim \in \{\leq, <, \geq, >\}$ and C, D are natural numbers.

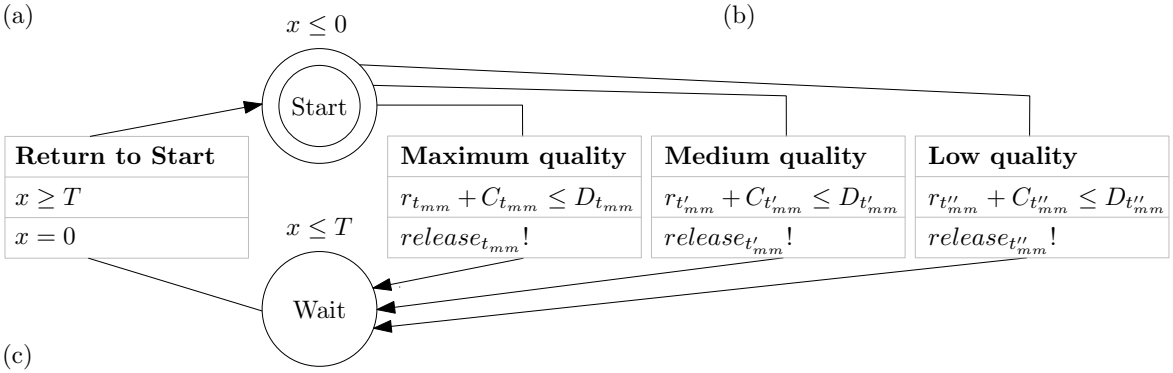
While a task automata can be checked for schedulability, the information about schedulability is not accessible within the task release automata in the previous work. In this work, we are presenting an encoding of the schedulability problem that enables the use of predicates for checking schedulability in edge guards.

	P	T	D	C	Description
t_{cl}	3	10	10	4	Call
t_{vc}	2	10	10	3	Video Chat
t_{mm}	1	10	10	7	Multimedia: max quality
t'_{mm}	1		10	4	Multimedia: medium quality
t''_{mm}	1		10	3	Multimedia: low quality

(a)



(b)



(c)

Figure 1: An Adaptive Smartphone System: (a) The task set; (b) Channel priorities that encode preference between the variants of the multimedia tasks; (c) A simplified task release automata for the multimedia task

Once the schedulability predicate has been added to a guard on the edge, the choice of the next state in automata can be determined by potential schedulability a system related to releases of different tasks. This novel encoding enabled us to model and verify systems that can conditionally release tasks according to the potential schedulability of those tasks.

3 An Example: An Adaptive Smartphone System

To demonstrate the idea behind our adaptive task release system, let us look at an example that would benefit from such a feature. Modern smartphone devices support multitasking and yet have quite limited resources available for realizing their functionality. We propose a solution that enables an idealized phone to adapt to the current situation on-the-fly by a dynamic restriction of a quality of service provided to the user.

The basic assumption is that the software in the smart phone is being executed in cycles. During each cycle, a series of short tasks that handle different applications are being executed. These tasks are presented in the Figure 1a. The applications that we have chosen for this example are: phone call, video call and multimedia. The user has the ability to turn on and off these applications at arbitrary moments in time. The status switch of the respective application will not be immediately reflected in the currently active task set, instead, the task set will change during the next cycle.

Tasks are described by four parameters: P - task priority, T - task period, D - relative deadline, C - required execution time. The multimedia application has three variants of the task corresponding to the three quality settings. Only the highest quality multimedia task has a period (t_{mm}), while the other two act as replacement in case the highest quality task cannot be executed.

The release model needs to choose a proper quality setting needs based on the amount of CPU time available while maintaining as high quality setting of the multimedia application as possible.

4 Graceful quality reduction

In order to meet all of the deadlines given in the Figure 1a, it is necessary to devise a method for graceful degradation of the quality of the multimedia application. Assuming a fixed priority scheduler, we model the scheduler and task queue as a timed automata inspired by the previous work [7]. In this model, we can observe the interference of higher priority tasks on any task.

To model the real-time task behavior, we need to keep track of the cumulative interference on some task t_i (by assigning it an integer variable r_i, and after releasing the task itself add its computation time to the value). This in turn lets us record the amount of computation time that has been dedicated to processing that interference and the task itself (a clock variable c_i), as well as how much time has elapsed from the release of task t_i, which is measured by the clock d_i. Clock d_i can be compared to the relative task deadline to get available time before the deadline. This approach has been previously investigated in a different context [6].

These variables are being updated throughout the simulation of the system, regardless of whether the respective task is actually released or not (except the case where the task would have the highest priority in the system, therefore entailing no interference). Assuming task t_i running, the question is, whether releasing task t_j of higher priority than t_i, would cause t_i to miss its deadline. The answer is given by evaluating the predicate $r_i - c_i + C_j \leq D_i - d_i$.

In the above predicate, $r_i - c_i$ represents the total required computation time to complete execution of the task t_i assuming all interferences that have been previously released. The right-hand side of the inequality, $D_i - d_i$, represents the leftover time to execute t_i before its deadline. Comparing these expressions gives us the amount of interference that can be added to t_i, without compromising its timely completion.

To address the problem described above, we have implemented a simplified version of this predicate on the lowest priority multimedia task. It chooses the maximum quality variant of the multimedia task that will still have time to execute as seen in Figure 1c. To ensure that we always choose the highest possible quality, we have established priorities between the synchronization channels Figure 1b that are used to release the tasks. Channel priorities are described in detail in [4].

References

[1] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In *Proc. of International Workshop on Formal Modeling and Analysis of Timed Systems*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[3] G. C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*. Kulwer Academic Publishers, 1997.

[4] Alexandre David, John Håkansson, Kim Larsen, and Paul Pettersson. Model checking timed automata with priorities using dbm subtraction. In Eugene Asarin and Patricia Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, volume 4202 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin / Heidelberg, 2006.

[5] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149 – 1172, 2007.

[6] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. theor. *Comput. Sci*, 354:301–317, 2006.

[7] Elena Fersman, Paul Pettersson, and Wang Yi. Timed automata with asynchronous processes: Schedulability and decidability. In *In Proceedings of TACAS 2002*, pages 67–82. Springer-Verlag, 2002.

Towards Integrated Modeling: Analytic Real-time Interfaces for Timed Automata based Component Models

Kai Lampka Lothar Thiele
Computer Engineering and Networks Lab, ETH Zurich, 8092 Zurich, Switzerland
{lampka,thiele}@tik.ee.ethz.ch

Abstract

For limiting component's mutual interference one may abstractly model interaction of components as streams of uniform, discrete activity-triggers, rather than using concrete data messages. Such event streams can abstractly be characterized by so called event arrival curves [10]. Moreover, the event arrival curves can be exploited for defining what kinds of traffic patterns a component is willing to accept and, on the level of outputs, defines what kind of stream the component guarantees to emit [11, 5]. The usage of such analytic, assume/guarantee (A/G), real-time interfaces features an integrated modelling environment as the formalisms describing the component models can be chosen freely, provided that a mapping from component models to the A/G interface is available. Moreover, this maintains compositionality as key properties of the overall system can be derived from the A/G interfaces rather than coping with the composed overall system at once. This extended abstract presents the machinery for deriving analytic A/G real-time interfaces based on arrival curves from Timed Automata (TA) [2] based component implementations.

1 Introduction

Motivation and Contribution. Today's real-time systems are embedded deeply into the physical environment and are highly integrated, thereby constituting complex systems. These systems have often to deliver safety-critical services, such that correctness of system-level designs is of uttermost importance. For coping with the complexity inherent to modern real-time systems, incremental, i.e., component-wise evolution of system designs, as well as integrated analysis frameworks appear to be an adequate mean. This extended abstract presents the machinery for deriving analytic real-time interfaces based on arrival curves of Real-Time Calculus (RTC) [10] from Timed Automata (TA) [2] based component implementations. This features (a) the component-wise evolution of systems and (b) provides an integrated framework for the analysis of heterogeneous system models. The reasons for this are as follows: as long as component models and interfaces are conformant, the interface derived properties of the overall systems are invariant w.r.t. refinement/substitution of components. In turn, the integrated modelling is realized by deriving analytic real-time interfaces based on arrival curves of RTC from TA-based component models. This allows us to handle analytic models, i.e.,models of the RTC, and operational models such as TA in a single framework, e.g. the Matlab-based MPA toolbox [1].
Finally it is also interesting to note that the usage of interfaces limits the complexity inherent to TA-based system analysis as state space explosion is limited to the level of component models, only experienced when deriving the analytic interfaces from the components.

Related Work. The authors of [6] established the foundation of interface theory, but explicitly excluded timed behavior from their elaboration. The theory on timed (state-based) interfaces was provided in [7]. The work presented in this paper elaborates on analytic, i.e.,state-less A/G real-time interfaces as established on the foundations of Real-time Calculus (RTC) [10] and presented in [11, 5]. The authors of [11] solely considered state-less RTC-based component models, adequate for a non-heterogeneous RTC-based system analysis. We also exploit the pattern presented in [8, 9] for coupling TA and RTC-based model descriptions.

2 Background Theory

Timed automata (TA). The level of component models is concerned with TA [2], where we employ the concepts as implemented in the timed model checker Uppaal, i.e.,Timed Safety Automata extended with variables [3, 4]. In the following we will also use the notation $\mathscr{M} \models \Phi$ which states that property Φ is true for TA $\mathscr{M}$. In the scope of this paper, it is sufficient to assume that Φ refers to some time-bounded reachability property, e. g. a bound on the size of a variable or a bound on a clock measuring the consumption and emission of a pair of events, signals respectively. Such safety properties, e. g. stated in Timed CTL (TCTL) are sufficient for verifying key performance such as buffer sizes, buffer underflows, and event-processing delays.

Streams and their abstract representation. A stream is a set of (potentially infinitely many) traces. It can be abstractly characterized by a tuple $\alpha := (\alpha^{low}, \alpha^{up})$ which is a pair of an upper and a lower arrival curve [10]. An arrival curve α bounds the number of events seen for any trace on the respective stream, i.e.,$\alpha^{low}(t-s) \leq R(t) - R(s) \leq \alpha^{up}(t-s)$ for all $0 \leq s \leq t$ and where R is the cumulative event counting function of the respective trace.
In the following we restrict the setting to arrival curves which can be composed from minimum and maximum operations on staircase curves: $\alpha^{up}(\Delta) := \min_{i \in I}(N_i + \left\lfloor \frac{\Delta}{\delta_i} \right\rfloor)$ and $\alpha^{low}(\Delta) := \max_{j \in J}(0, N_j + \left\lfloor \frac{\Delta}{\delta_j} \right\rfloor)$ where we assume that α^{up} has increasing step widths, and α^{low} possess decreasing staircase widths. This models what we call a pseudo-concave/convex curve. How to implement an arrival curve of the above kind by a set of cooperating TA is discussed in [8, 9]. Such a sets of TA is denoted in the following as input generator $\mathscr{G}$. It is capable of emitting all traces of a dedicated event type e and w. r. t. bounding curve α. We use the notation $\mathscr{G}(\alpha)$ or if necessary $\mathscr{G}(\alpha^{up}, \alpha^{low})$ for emphasizing this. As each event e emitted by $\mathscr{G}$ may trigger subsequent behaviour in a down-streamed, user-defined TA-based component model $\mathscr{M}$ we also speak of stimulated component models. For the composite consisting of the input generator $\mathscr{G}$ and the user-defined component model $\mathscr{M}$ to be stimulated we employ the notation $\mathscr{G} \| \mathscr{M}$. Executing a stimulated component model $\mathscr{G} \| \mathscr{M}$ together with a set of observer TA $\mathscr{O}$ in a binary search allows one to obtain a set of parameters. These parameters can be used for constructing a pseudo-concave/convex curve bounding the (output) stream emitted by $\mathscr{G} \| \mathscr{M}$.

3 Analytic real-time interfaces for TA-based component models

(A) Definitions. A (single port) component $\mathscr{C}$ is a triple $(In, \mathscr{M}, Out)$, where In is the input port for consuming event-triggers of a dedicated type, $\mathscr{M}$ is a component model which consumes and emits events of a dedicated type, and Out is the dedicated output port for emitting event-triggers of a dedicated type. An analytic (single port) A/G Interfaces $\mathscr{I}$ is a triple $(\Phi, \alpha_{in}^{\mathscr{I}}, \alpha_{out}^{\mathscr{I}})$, where Φ is a set of dedicated (safety) properties invariantly fulfilled by any component implementing the interface. $\alpha_{in}^{\mathscr{I}}$ is the input bound which is a RTC curve bounding the cumulative counting function of any stream of event-triggers fed into component $\mathscr{M}$. $\alpha_{out}^{\mathscr{I}}$ is the output bound which is a RTC curve bounding the cumulative counting function of any stream of event-triggers emitted by component $\mathscr{M}$.

Formally, we relate interface definitions and components models now as follows: Let $\mathscr{G}$ be an input generator for restricting the input (streaming) behaviour of the environment. A TA-based component $\mathscr{M}$ implements an interface $\mathscr{I} := (\Phi, \alpha_{in}^{\mathscr{I}}, \alpha_{out}^{\mathscr{I}})$ if the following conditions apply:

(a) $\mathscr{M}$ and $\mathscr{I}$ are event-compatible.

(b) All traces tr produced by composite $\mathscr{G}(\alpha_{in}^{\mathscr{I}}) \| \mathscr{M}$ and filtered w. r. t. an dedicated output event, e. g. of type o, are bounded by arrival curve $\alpha_{out}^{\mathscr{I}}$.

(c) For each reachable system states s of composite $\mathscr{G}(\alpha_{in}^{\mathscr{I}}) \| \mathscr{M}$ it holds: $s \models \Phi$.

If $\mathscr{M}$ implements $\mathscr{I}$ we speak of **conformance** of component $\mathscr{M}$ and interface $\mathscr{I}$. Verifying if $\mathscr{M} \lhd \mathscr{I}$ applies is straight-forward: one encodes the input/output curves of the interface and executes the approach of [8, 9] which allows one to decide if $\mathscr{G} \| \mathscr{M} \| \mathscr{O} \models \Phi$ holds or not.

Alternatively, one may execute a binary search for either computing the bound on a component's input stream or on its output stream. This features the definition of an analytic A/G interface for TA-based component models and will be introduced now; an illustration of the scheme is provided by Fig.1.

(B) Scheme for deriving the input interface. The basic scheme for computing a conservative and pseudo-concave/convex bound for a component's input streams, i.e.,the detection of and $\alpha_{in}^{\mathscr{I}}$ can be organized as binary search. In this search one changes the parameters of the staircase curves approximating the upper arrival curve $\alpha_{in}^{\mathscr{I}}$ and queries a timed model checker, e.g. Uppaal, if $\mathscr{G}(\alpha_{in}^{\mathscr{I}}) \| \mathscr{M} \| \mathscr{O}(\alpha_{out}^{\mathscr{I}}) \models \Phi$ holds. One may recall that each of the staircase curves is defined by its (burst-)capacity N_i and step-width δ_i. The procedure in finding these parameters can be partitioned in two steps.

Bounding the long-term behaviour of inputs. By employing a timed model checker, e.g. Uppaal, in a binary search one may compute the smallest step-width δ s.t. $\mathscr{G}(\alpha_{in}^{up}) \| \mathscr{M} \| \mathscr{O}(\alpha_{out}^{\mathscr{I}}) \models \Phi$ holds. At termination the search has found the "steepest" long term rate d_{LT} acceptable to the component model. As input the algorithm requires some safe bounds $(\delta_{up}, \delta_{low})$, which are the smallest and largest values assigned to search parameter δ_{crr}. With δ fixed to d_{LT} one is now enabled to execute a binary search for binding parameter N. Such a search yield the largest value B_{LT} s.t. for $\alpha_{in}^{up}(\Delta) := B_{LT} + \lfloor \frac{\Delta}{d_{LT}} \rfloor \, \mathscr{G}(\alpha_{in}) \| \mathscr{M} \| \mathscr{O}(\alpha_{out}^{\mathscr{I}}) \models \Phi$ holds. In the same way one proceeds for finding a lower curve α_{in}^{low}. At termination this scheme yields an approximation for an input curve $\alpha_{in} := (\alpha_{in}^{up}, \alpha_{in}^{low})$. This input curve can now be refined, particularly with respect to the short-term streaming behaviour. For conciseness we focus on the tightening of an upper input bound α_{in}^{up}, where we model α_{in}^{up} as minimum of two staircase curves.

Bounding the short-term behaviour of inputs. When dealing with more than one staircase curves one faces a degree of freedom w.r.t. the choice of the free parameters to be queried in the binary search, here (burst-)parameters N_1, N_2 and step-width δ_1, parameter δ_2 is bound as it is the long term rate d_{LT}. When choosing appropriate values for the free parameters the following relation holds $B_{LT} \leq N_1 \leq N_2$ and $0 < \delta_1 \leq d_{LT}$; this is because we model α_{in}^{up} as minimum of two staircase curves. According to this a possible initialization could than be $N_1 := B_{LT}, N_2 := k \cdot B_{LT}$ and carry out a binary search for determining δ_1. Inadequate choices are detectable, as the search terminates and $\delta_1 = d_{LT}$ holds. In such cases the search has to be restarted, but at least with a smaller value for N_2.

One may note that the constructed curve α_{in} is a conservative bound as $\alpha_{in}(s-t) \leq \alpha^{pot}(s-t)$ for all $0 \leq s < t$ holds, where with α^{pot} as the maximal input curve which is unknown.

Fig. 1 illustrates the procedure for approx. an unknown input bound α^{pot} with two staircase segments; for illustration purpose Fig. 1 abstracts away that we are actually dealing with staircase functions. At first one searches for the steepest long term rate which is depicted in Fig.1(a), where δ_{crr} is the current slope to be tested in the binary search. At termination the scheme delivers a value d_{LT}. With δ_{crr} fixed we search now for the largest value for N, which gives us the input (burst-)parameter B_{LT} (Fig. 1(b)). With the long-term slope d_{LT} and the input (burst-)parameter B_{LT} one proceeds with the routine for finding a tighter bound on the short term streaming behaviour (Fig.1(c) and 1(d)). As shown in Fig. 1(c) the initial choice for N_2 was to large, hence the search for finding an adequate δ_1 stops unsuccessfully, namely once $\delta_1 = d_{LT}$. In a second run which uses a smaller value for parameter N_2 the finding of δ_1 terminates once $\mathscr{G} \| \mathscr{M} \| \mathscr{O} \models \Phi$ holds.

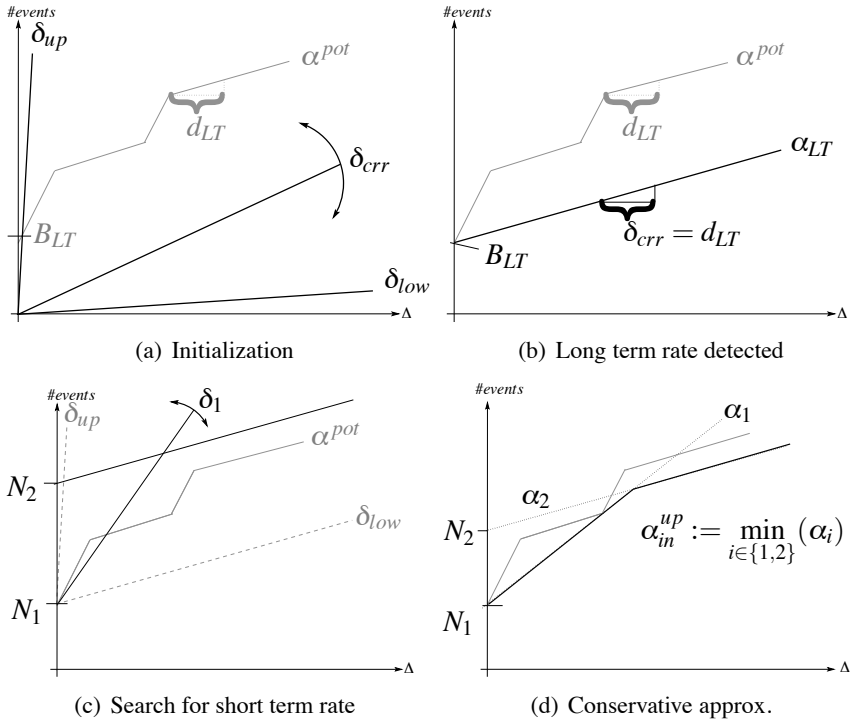


Figure 1: Finding an input bound for a TA-based component model

(C) Scheme for deriving the output interface. The input bound is fixed $\alpha_{in}^{\mathscr{I}}$, allowing one to execute a set of observer TA $\mathscr{O}(\alpha)$ together with composite $\mathscr{G}(\alpha_{in}^{\mathscr{I}})\|\mathscr{M}$ and property Φ in a binary search. The obtained parameters allow one to construct a pseudo-concave/convex output bound [8, 9].

References

[1] Modular Performance Analysis Framework and Matlab Toolbox. www.mpa.ethz.ch.

[2] R. Alur and D. L. Dill. Automata For Modeling Real-Time Systems. In M. Paterson, editor, *Proc. of ICALP 1990*, volume 443 of *LNCS*, pages 322–335. Springer, 1990.

[3] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[4] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004.

[5] S. Chakraborty, Y. Liu, N. Stoimenov, L. Thiele, and E. Wandeler. Interface-based rate analysis of embedded systems. In *RTSS 2006*, pages 25–34, 2006.

[6] L. de Alfaro and T. A. Henzinger. Interface theories for component-based design. In T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT 2001*, volume 2211 of *LNCS*, pages 148–165. Springer, 2001.

[7] L. de Alfaro, T. A. Henzinger, and M. I. A. Stoelinga. Timed interfaces. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *EMSOFT 2002*, LNCS, pages 108–122. Springer, 2002.

[8] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. In *EMSOFT 2009*, pages 107–116. ACM/IEEE, 2009.

[9] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: A hybrid methodology for the performance analysis of embedded real-time systems. *Design Automation for Embedded Systems*, 14(3):193–227, 2010.

[10] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. Int. Symposium on Circuits and Systems*, volume 4, pages 101–104, 2000.

[11] E. Wandeler and L. Thiele. Real-time interfaces for interface-based design of real-time systems with fixed priority scheduling. In *EMSOFT 2005*, pages 80–89. ACM/IEEE, 2005.

A Mode Switch Logic for component-based multi-mode systems

Yin Hang
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
young.hang.yin@mdh.se

Hans Hansson
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
hans.hansson@mdh.se

Abstract

Component-Based Development (CBD) reduces development time and effort by allowing systems to be built from pre-developed reusable components. A classical approach to reduce embedded systems design and run-time complexity is to partition the behavior into a set of major system modes. In supporting system modes in CBD, a key issue is seamless composition of multi-mode components into systems. In addressing this issue, we have developed a Mode Switch Logic (MSL) for component-based multi-mode systems, implementing seamless coordination and synchronization of mode switch in systems composed of independently developed components.

1 Introduction

Traditionally, partitioning system behaviors into different operational modes has been used to reduce complexity and improve resource efficiency. Each mode corresponds to a specific system behavior. The system can start by running in a default mode and switches to another appropriate mode when some condition changes. In this way, the complexity of both system design and verification can be reduced while system execution efficiency is improved. A typical multi-mode system is the control software of an airplane, which e.g. could run in *taxi* mode (the initial mode), *taking off* mode, *flight* mode and *landing* mode.

There are a variety of alternatives to design and develop a multi-mode system. We set our focus on Component-Based Development (CBD), a promising solution for the development of embedded systems. The most adorable feature of CBD for us is its component reuse idea, which allows us to build a system by reusable components, i.e. a system does not have to be developed from scratch, instead, some of its components or subsystems may be directly obtained from a repository of pre-developed components.

Our target is component-based multi-mode systems (CBMMSs), i.e. multi-mode systems built by a set of hierarchically organized components. Figure 1 illustrates a simple CBMMS. From the top level, the system consists of Component a (which is further decomposed into Component c, d and e) and b (which is further decomposed into Component f and g). The system and its components all support two modes: M_1 and M_2. In mode M_1, Component g is deactivated (invisible) and Component f has one mode-specific behavior (indicated by black color). In mode M_2, Component g becomes activated while Component d is deactivated. Besides, Component f has another mode-specific behavior (indicated by grey color). Component connection is presented in Figure 1(b). Due to the availability/unavailability of components d and g, component connections are different in these two modes.

For a CBMMS, the system behavior is highly dependent on its components. This dependency also holds during a mode switch. When a system switches from one mode to another mode, its mode switch is essentially achieved by the mode switches of certain components. The central issue is that the mode switches of different components must be coordinated and synchronized to achieve a successful and efficient global mode switch. Notwithstanding that there is plenty of research work dealing with mode switch, little attention has been paid to this composable mode switch problem. To this end, we have developed a Mode Switch Logic (MSL) for CBMMSs [1].

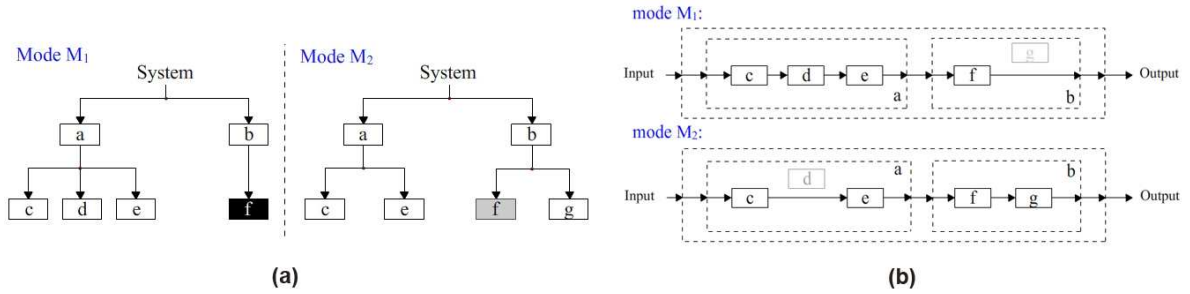


Figure 1: A component-based multi-mode system

2 The Mode Switch Logic (MSL)

Our MSL includes three major facets: (1) component redefinition and configuration, (2) MSR (Mode Switch Request) propagation mechanism, and (3) mode switch dependency rules. Using our MSL, mode switch support is added to each component. Based on traditional components, which do not support multiple modes, we introduce specific ports dedicated to mode switch, define component configuration for each mode and integrate the MSL into each component. The mode switch related communication between different components is realized by communication over those dedicated ports. Some components may reconfigure themselves during mode switch and this is controlled by the MSL of each component. Component configuration varies between primitive and composite components. For primitive components, component configuration is defined by the component running status (activated/deactivated) and the mode-specific behavior. For composite components, the component configuration is defined by the component running status, the activated subcomponents and the active inner component connections. In each mode, only activated components are running, while deactivated components are temporarily unavailable. Likewise, in each mode, only active component connections are considered. A connection becomes inactive when it is disconnected due to a mode switch. When a component starts its mode switch, it will reconfigure itself by changing the aforementioned elements.

The MSR propagation mechanism ensures that when a component triggers a mode switch, all the related components can be notified as soon as possible. *MSR* is a signal telling each component to switch mode. It is originally triggered by a particular component and then propagated to all related components. Our MSR propagation mechanism guarantees that all the components are notified by the same *MSR*, and that it avoids any potential redundant *MSR* transmission.

The mode switch dependency rule specifies which component(s) should complete mode switch before which other component(s), based on the component hierarchy and component connections. Usually different dependency rules are suitable for different types of component connections. Due to the dependency rule, a component may not proceed with its mode switch until particular conditions are satisfied. In [1], we present a forward dependency rule for ”pipe-and-filter” systems and implement both the MSR propagation mechanism and the forward dependency rule as algorithms for primitive and composite components.

For a multi-mode real-time system, it’s important to analyze its mode switch time. In [2], we provide the mode switch timing analysis based on our MSL. Different from most existing works which target tasks, our target is component. We divide the global mode switch of a system into three phases: MSR propagation phase, component reconfiguration phase, and mode switch completion phase. The total mode switch time is equivalent to the sum of the duration in these three phases.

The correctness of our MSL has been verified by model-checking using UPPAAL [3], with regard to both functional and timing aspects.

3 Current work and future work

Our original MSL used to be constrained by an unrealistic assumption that all the components support the same modes. In our current work, we propose a mode mapping mechanism which can be directly implemented during MSR propagation so that our MSR propagation mechanism still works when different components support different modes. Another issue that we will consider is atomic component executions. Our current assumption is that the execution of any component can be immediately interrupted by a *MSR*, however, in a real system it could be possible that the execution of one component or even a group of components has to reach a specific "stable" state before it can be interrupted. Such atomic component execution can increase the global mode switch latency, thus it plays an important role in mode switch timing analysis. Furthermore, when our MSL is mature enough, it is our ambition to implement it in the ProCom framework [4] that embodies the feature of component reuse very well.

4 Acknowledgments

This work is supported by the Swedish Research Council.

References

[1] Y. Hang, E. Borde, and H. Hansson. Composable mode switch for component-based systems. In *APRES '11: Third International Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 19–22, 2011.

[2] Y. Hang and H. Hansson. Timing analysis for a composable mode switch. In *The Work-in-Progress session of the 23rd Euromicro Conference on Real-Time Systems*, pages 15–18, 2011.

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

[4] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson. Formal semantics of the ProCom real-time component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 478–485, 2009.

A Categorical View of Bisimulation for Higher Dimensional Automata*

Elena Oshevskaya[1], Irina Virbitskaite[1,2], Eike Best[3]

[1] A.P. Ershov Institute of Informatics Systems, SB RAS
[2] Novosibirsk State University
[3] Carl von Ossietzky University of Oldenburg

1 Introduction

In order to unify and clarify apparent differences between the extensive amount of research within the field of behavioral equivalences, several category-theoretic approaches to the matter have appeared. Two of them were initiated by Joyal, Nielsen, and Winskel in [2] where they have proposed abstract ways of capturing the notion of behavioral equivalence through open maps based bisimilarity and its logical counterpart — path bisimilarity. Another way to provide categorical characterizations is to adopt the coalgebraic approach. One of the basic strands of the research is concerned with a coalgebraic rendering of various behavioral equivalences in the linear time – branching time spectrum.

The most popular geometric model for concurrency is higher dimensional automata (HDA) which have been proposed by V. Pratt [4]. As shown in the paper [1], higher dimensional automata are more expressive than most of the truly concurrent models taking branching time fully into account (e.g., Petri nets, event structures). The contribution of the paper is to show how several categorical (open maps, path-bisimilarity and coalgebraic) approaches to an abstract characterization of bisimulation relate to each other and to hereditary history preserving bisimulation, in the setting of HDA.

2 Higher Dimensional Automata

The following is the well known definition of HDA.

A *precubical set* M is a collection of sets $(M_n)_{n\in\mathbb{N}}$, such that $M_n \cap M_k = \emptyset$ for all $n \neq k$, together with boundary functions $M_n \overset{d_i^0}{\underset{d_j^1}{\rightrightarrows}} M_{n-1}$ for all $n \in \mathbb{N}$ and $i, j = 1 \ldots n$, such that $d_i^k \circ d_j^m = d_{j-1}^m \circ d_i^k$ $(i < j,\ k, m = 0, 1)$.

A *(labelled) HDA* (*over a set L of actions*) is a triple $\mathrm{M} = (M, i_0^{\mathrm{M}}, l_L^{\mathrm{M}})$, where M is a precubical set, $i_0^{\mathrm{M}} \in M_0$ is a distinguished basepoint of M, called the *initial point*, $l_L^{\mathrm{M}} : M_1 \to L$ is a *labelling function* from the 1-cubes of M to a set L of actions such that $l_L^{\mathrm{M}}(d_i^0(x)) = l_L^{\mathrm{M}}(d_i^1(x))$ for all $i = 1, 2$ and $x \in M_2$.

Let $\mathrm{M} = (M, i_0^{\mathrm{M}}, l_{L^{\mathrm{M}}}^{\mathrm{M}})$ and $\mathrm{N} = (N, i_0^{\mathrm{N}}, l_{L^{\mathrm{N}}}^{\mathrm{N}})$ be HDA. A mapping $\mathrm{f} = \langle f, \alpha\rangle$ (where $f = \cup f_n$, $f_n : M_n \to N_n$ and $\alpha : L^{\mathrm{M}} \to L^{\mathrm{N}}$) is called a *morphism* from M to N iff it holds: (1) $f_0(i_0^{\mathrm{M}}) = i_0^{\mathrm{N}}$, (2) $l_{L^{\mathrm{N}}}^{\mathrm{N}} \circ f = \alpha \circ l_{L^{\mathrm{M}}}^{\mathrm{M}}$, (3) $f_n \circ d_i^m = d_i^m \circ f_{n+1}$. HDA with morphisms between them form a category **HDA** in which the composition of two morphisms $\mathrm{f} = \langle f, \alpha\rangle : \mathrm{M} \to \mathrm{M}'$ and $\mathrm{g} = \langle g, \beta\rangle : \mathrm{M}' \to \mathrm{M}''$ is $\mathrm{g} \circ \mathrm{f} = \langle g \circ f,\ \beta \circ \alpha\rangle : \mathrm{M} \to \mathrm{M}''$, and the identity morphism is a pair of the identity functions.

3 Hereditary history preserving bisimulation

In order to reason about the behaviour of HDA, we introduce the following notions and notations. A *cubical path* in an HDA M over L is a sequence $P = p_0 \xrightarrow{d_{j_1}^{m_1}} \ldots \xrightarrow{d_{j_k}^{m_k}} p_k$[1] of cubes and boundary functions, such that $d_{j_s}^{m_s}(p_s) = p_{s-1}$, if $m_s = 0$, and $d_{j_s}^{m_s}(p_{s-1}) = p_s$, if $m_s = 1$, for all $1 \leq s \leq k$, and, moreover, $p_0 = i_0^{\mathrm{M}}$. For cubical paths $P = p_0 \xrightarrow{d_{j_1}^{m_1^p}} \ldots \xrightarrow{d_{j_k}^{m_k^p}} p_k$ and

*This work is supported in part by the DFG-RFBR (grant No 436 RUS 113/1002/01, 09-01-91334). The first author is partly supported by the RFBR (grant 09-01-00598-a) and by the President Program "Leading Scientific Schools" (grant NSh-7256.2010.1).

[1] We write $P = p_0 p_1 \ldots p_k$ if no confusion arises.

$Q = q_0 \xrightarrow[j_1^q]{d_1^{m_1^q}} \ldots \xrightarrow[j_n^q]{d_n^{m_n^q}} q_n$, we say that Q is an extension of P (denoted $P \to Q$) if $n \geq k$ and $p_s = q_s$, $m_s^p = m_s^q$, $j_s^p = j_s^q$ for all $0 \leq s \leq k$. In particular, we write $P \xrightarrow{d_j^m} Q$ if $n = k+1$, $m_{k+1}^q = m$ and $j_{k+1}^q = j$.

A *homotopy* (denote $\sim$) is the least equivalence on cubical paths in M such that if P and P' are *s-adjacent* (denote $P \overset{s}{\leftrightarrow} P'$), i.e. P' can be obtained from P by replacing (for $i < j$ and $m = 0, 1$) either a segment $\xrightarrow{d_i^0} p_s \xrightarrow{d_j^m}$ by a segment $\xrightarrow{d_{j-1}^m} p'_s \xrightarrow{d_i^0}$, or vice versa; or a segment $\xrightarrow{d_j^m} p_s \xrightarrow{d_i^1}$ by a segment $\xrightarrow{d_i^1} p'_s \xrightarrow{d_{j-1}^m}$, or vice versa, then P and P' are equivalent. Moreover, P and P' are (s,u,v)-*adjacent* (denote $P \overset{(s,u,v)}{\longleftrightarrow} P'$), if P' can be obtained from $P = \hat{p}_0 \ldots \hat{p}_s \ldots \hat{p}_k$ by an s-adjacency replacement of the segment $\xrightarrow{d_u^n} \hat{p}_s \xrightarrow{d_v^l}$.

Define a behavioural equivalence on HDA, called hereditary history preserving bisimulation (hhp-bisimulation).

Definition 1. Let M and N be HDA over L.

Cubical paths $P = p_0 \ldots p_k$ in M and $Q = q_0 \ldots q_k$ in N are called *l-related* iff $l^{\mathrm{M}}(p_s) = l^{\mathrm{N}}(q_s)$ for all $0 \leq s \leq k$.

A binary relation $\mathcal{R}$ on cubical paths in M and N is called a *hereditary history preserving bisimulation (hhp-bisimulation)* between M and N if for any $(P, Q) \in \mathcal{R}$, P and Q are l-related and the following conditions are satisfied:

1. if $P \xrightarrow{d_i^m} P'$ then $Q \xrightarrow{d_i^m} Q'$ and $(P', Q') \in \mathcal{R}$ for some Q' in N, and vice versa,
2. if $P' \to P$ then $Q' \to Q$ and $(P', Q') \in \mathcal{R}$ for some Q' in N, and vice versa,
3. if $P \overset{(s,u,v)}{\longleftrightarrow} P'$ then $Q \overset{(s,u,v)}{\longleftrightarrow} Q'$ and $(P', Q') \in \mathcal{R}$ for some Q' in N, and vice versa.

HDA M and N are *hhp-bisimilar* if there exists an hhp-bisimulation between them which relates their initial points (regarded as cubical paths).

4 Open Maps Bisimulation

In this section, we establish the coincidence between hhp-bisimulation and open maps based bisimulation in the context of HDA.

We consider **HDA** as a category of models. For our purpose, we need to endow **HDA** with a fibred structure. Let $\mathbf{HDA}_L$ denote the subcategory of **HDA** whose objects are HDA labelled over L and morphisms have the identity action component. In the category **HDA**, we have to choose a full subcategory of path objects. For a natural number N, define the N-cube $\boxplus^N$ as $\{0\}$, if $N = 0$, and $\{(t_1, \ldots, t_N) \mid t_j \in \{0, \frac{1}{2}, 1\}\}$, otherwise. Clearly, the N-cube $\boxplus^N$ can be split into the sets $(\boxplus^N)_n = \{(t_1, \ldots, t_N) \in \boxplus^N \mid |\{t_j = \frac{1}{2} \mid 1 \leq j \leq N\}| = n\}$, where $0 \leq n \leq N$. For $(t_1, \ldots, t_N) \in \boxplus^N_n$, let the indexes $j_1 < \ldots < j_n$ be such that $t_{j_i} = \frac{1}{2}$ for all $1 \leq i \leq n$. Determine the boundary functions $\widetilde{d}_i^m : (\boxplus^N)_n \to (\boxplus^N)_{n-1}$ as follows: $\widetilde{d}_i^m(t_1, \ldots, t_N) = (t_1, \ldots, t_{j_i - 1}, m, t_{j_i+1} \ldots, t_N)$, for all $m = 0, 1$, $1 \leq i \leq n$, and $0 < n \leq N$. Obviously, $\boxplus^N$ is a precubical set. Construct an HDA $\boxplus^{\mathrm{N}}$ over L as follows: $\boxplus^{\mathrm{N}} = (\boxplus^0, 0, \emptyset)$, if $N = 0$, and $\boxplus^{\mathrm{N}} = (\boxplus^N, (\underbrace{0, \ldots, 0}_{N}), l)$, otherwise (here, l is a labelling function from $(\boxplus^N)_1$ to a set L of actions, satisfying $l(\widetilde{d}_i^0(x)) = l(\widetilde{d}_i^1(x))$ for all $i = 1, 2$ and $x \in (\boxplus^N)_2$). A *path object* is an HDA having the form of a cubical path $\widetilde{P} \in \mathcal{CP}_p(\boxplus^{\mathrm{N}})$ such that $\widetilde{d}_1^1 \circ \cdots \circ \widetilde{d}_{\dim p}^1(p) = (\underbrace{1, \ldots, 1}_{N})$, if $N > 0$. We use **cP** to denote the full subcategory of the category **HDA**, whose objects are path objects. Clearly, the category $\mathbf{cP}_L$ is small, for a given set L of actions.

Theorem 1. *Two objects in* $\mathbf{HDA}_L$ *are* $\mathbf{cP}_L$*-bisimular iff they are hhp-bisimular.*

5 Path Bisimulation

To obtain a logic characteristic of bisimulation induced by open maps, Joyal, Nielsen, and Winskel [2] have proposed a second category-theoretic characterization of bisimulation — path bisimulation which is a relation based generalization of open maps bisimulation.

Definition 2. Let $\mathbb{M}$ be a category of models, let $\mathbb{P}$ be a small category of path objects, where $\mathbb{P}$ is a subcategory in $\mathbb{M}$, let I be a common initial object[2] in $\mathbb{P}$ and $\mathbb{M}$. Then,

- Two objects X_1 and X_2 in $\mathbb{M}$ are called *path-$\mathbb{P}$-bisimilar* iff there is a set $\mathcal{B}$ of pairs of paths (p_1, p_2) with common domain P, so $p_1 : P \to X_1$ is a path in X_1 and $p_2 : P \to X_2$ is a path in X_2, such that
  - (o) $(i_1, i_2) \in \mathcal{B}$, where $i_1 : I \to X_1$ and $i_2 : I \to X_2$ are the unique paths starting in the initial object, and for all

[2] In the case when $\mathbb{P}$ is $\mathbf{cP}_L$ and $\mathbb{M}$ is $\mathbf{HDA}_L$, the initial object is the HDA $\boxplus^0$.

$(p_1, p_2) \in \mathcal{B}$ and for all $m : P \to Q$ in $\mathbb{P}$, it holds:

(i) if there exists $q_1 : Q \to X_1$ with $q_1 \circ m = p_1$ then there exists $q_2 : Q \to X_2$ with $q_2 \circ m = p_2$ and $(q_1, q_2) \in \mathcal{B}$ and

(ii) if there exists $q_2 : Q \to X_2$ with $q_2 \circ m = p_2$ then there exists $q_1 : Q \to X_1$ with $q_1 \circ m = p_1$ and $(q_1, q_2) \in \mathcal{B}$.

- Two objects X_1 and X_2 in $\mathbb{M}$ are *strong path-$\mathbb{P}$-bisimilar* iff they are path-$\mathbb{P}$-bisimilar and the set $\mathcal{B}$ further satisfies:

(iii) If $(q_1, q_2) \in \mathcal{B}$, with $q_1 : Q \to X_1$ and $q_2 : Q \to X_2$ and $m : P \to Q$ in $\mathbb{P}$, then $(q_1 \circ m, q_2 \circ m) \in \mathcal{B}$.

Theorem 2. *Two objects in* $\mathbf{HDA}_L$ *are strong path-*$\mathbf{cP}_L$*-bisimular iff they are hhp-bisimular.*

6 Coalgebraic Bisimulation

Another alternative abstract characterization of bisimulation is based on a category of coalgebras induced by an endofunctor on an arbitrary category.

We start with defining the terminology from [3]. Let $\mathbb{M}$ be a locally small category with a small path subcategory $\mathbb{P}$. We will define an embedding of $\mathbb{M}$ into a category of coalgebras for some endofunctor on the category $Set^{|\mathbb{P}|}$ of $|\mathbb{P}|$-sorted sets ($|\mathbb{P}|$-indexed sets), where $|\mathbb{P}|$ is the set of objects in $\mathbb{P}$. The endofunctor $F_{\mathbb{P}} : Set^{|\mathbb{P}|} \longrightarrow Set^{|\mathbb{P}|}$ is defined as follows:

$$\{X_P\}_{P\in|\mathbb{P}|} \longmapsto \{\prod_{Q\in|\mathbb{P}|} (\mathcal{P}(X_Q))^{Hom_{\mathbb{P}}(P,Q)}\}_{P\in|\mathbb{P}|},$$

where $\mathcal{P}(\cdot)$ denotes the powerset, X_P specifies a component of a $|\mathbb{P}|$-sorted set X for $P \in |\mathbb{P}|$, and $Hom_{\mathbb{P}}(P, Q)$ stands for the set of all morphisms from P to Q in $\mathbb{P}$. On morphisms in the category $Set^{|\mathbb{P}|}$, $F_{\mathbb{P}}$ acts by the following rule:

$$F_{\mathbb{P}} : (\{\gamma_P\}_{P\in|\mathbb{P}|} : X \to Y) \longmapsto \{\prod_{Q\in|\mathbb{P}|} h^P_Q\}_{P\in|\mathbb{P}|},$$

where $h^P_Q : \mathcal{P}(X_Q)^{Hom_{\mathbb{P}}(P,Q)} \to \mathcal{P}(Y_Q)^{Hom_{\mathbb{P}}(P,Q)} : g \longmapsto f$, $f(m) = \{\gamma_Q(x) \mid x \in g(m)\}$ for all $m \in Hom_{\mathbb{P}}(P, Q)$.

A *coalgebra for* $F_{\mathbb{P}}$ or *$F_{\mathbb{P}}$-coalgebra* is a pair (S, tr) with S an object in $Set^{|\mathbb{P}|}$ and $tr : S \to F_{\mathbb{P}}(S)$ a morphism in $Set^{|\mathbb{P}|}$, which consists of a family of functions:

$$\{tr_P : S_P \to \prod_{Q\in|\mathbb{P}|} (\mathcal{P}(S_Q))^{Hom_{\mathbb{P}}(P,Q)}\}_{P\in|\mathbb{P}|}.$$

The set S is called the *carrier* and the function tr is called the *coalgebra structure* of the $F_{\mathbb{P}}$-coalgebra. A morphism $\gamma : S_1 \to S_2$ in the category $Set^{|\mathbb{P}|}$ is called a *cohomomorphism* between $F_{\mathbb{P}}$-coalgebras (S_1, tr_1) and (S_2, tr_2) iff $F_{\mathbb{P}}(\gamma) \circ tr_1 = tr_2 \circ \gamma$. $F_{\mathbb{P}}$-coalgebras and cohomomorphisms between them constitute a category, denoted by $\mathcal{CA}_{\mathbb{P}}$.

An *$F_{\mathbb{P}}$-bisimulation* between two coalgebras (S_1, tr_1) and (S_2, tr_2) is a $|\mathbb{P}|$-sorted relation $R = \{R_P\}_{P\in|\mathbb{P}|} \subseteq (S_1 \times S_2)$ such that, whenever $(m_1, m_2) \in R_P$ and $m : P \to Q$ in $\mathbb{P}$, then (i) if $m_1 \xrightarrow{m} m_1'$, then $m_2 \xrightarrow{m} m_2'$ and $(m_1', m_2') \in R_Q$ for some $m_2' \in S_2$, and (ii) vice versa. Here, for an *$F_{\mathbb{P}}$-coalgebra* (S, tr), $m_1 \xrightarrow{m} m_2$ denotes a triple $\langle m_1, m, m_2\rangle$ satisfying $m_2 \in tr_P(m_1)(m)$.

Next, following [3], we relax the requirement on coalgebra morphism. A morphism $\gamma : S \to S'$ in $Set^{|\mathbb{P}|}$ is called a *lax cohomomorphism* between $F_{\mathbb{P}}$-coalgebras (S, tr) and (S', tr') if for each $s \in S_P$ and $m \in Hom_{\mathbb{P}}(P, Q)$, $\{\gamma_Q(r) \mid r \in tr_P(s)(m)\} \subseteq tr'_P(\gamma_P(s))(m)$. $F_{\mathbb{P}}$-coalgebras and lax cohomomorphisms constitute a category, denoted by $\mathcal{CA}^{lax}_{\mathbb{P}}$. For $\mathbb{M}$ with $\mathbb{P}$, recall the functor $\mathcal{B}eh^{\mathbb{M}}_{\mathbb{P}} : \mathbb{M} \to \mathcal{CA}^{lax}_{\mathbb{P}}$ from [3]. $\mathcal{B}eh^{\mathbb{M}}_{\mathbb{P}}$ acts on objects X in $\mathbb{M}$ as follows: $\{Hom_{\mathbb{M}}(P, X)\}_{P\in|\mathbb{P}|}$ is the carrier and $\{tr_P : m_1 \longmapsto \prod_{m\in\uplus_{Q\in|\mathbb{P}|}Hom_{\mathbb{P}}(P,Q)}\{m_2 \mid m_1 = m_2 \circ m\})\}_{P\in|\mathbb{P}|}$ is the coalgebra structure of the corresponding $F_{\mathbb{P}}$-coalgebra. $\mathcal{B}eh^{\mathbb{M}}_{\mathbb{P}}$ acts on morphisms $f : X \to Y$ in $\mathbb{M}$ as follows: $\mathcal{B}eh^{\mathbb{M}}_{\mathbb{P}}(f)_P : Hom_{\mathbb{M}}(P, X) \to Hom_{\mathbb{M}}(P, Y) : \alpha \longmapsto (f \circ \alpha)$.

Proposition 1. *For any two objects* M *and* M′ *in* $\mathbf{HDA}_L$, $\mathbf{cP}_L$*-bisimulation implies path-*$\mathbf{cP}_L$*-bisimulation coinciding with* $F_{\mathbf{cP}_L}$*-bisimulation between* $\mathcal{B}eh^{\mathbf{HDA}_L}_{\mathbf{cP}_L}(\mathrm{M})$ *and* $\mathcal{B}eh^{\mathbf{HDA}_L}_{\mathbf{cP}_L}(\mathrm{M}')$*, containing the pair* $(i_{\mathrm{M}}, i_{\mathrm{M}'})$*, where* $i_{\mathrm{M}} : \boxplus^0 \to \mathrm{M}$ *and* $i_{\mathrm{M}'} : \boxplus^0 \to \mathrm{M}'$ *are paths, with the initial object* $\boxplus^0$*.*

References

[1] van Glabbeek, R.J.: On the Expressiveness of higher dimensional automata. Theor. Comput. Sci. 356 (3), 265–290 (2006)

[2] Joyal, A., Nielsen, M., Winskel, G.: Bisimulation from open maps. Inform. Comput. 127(2), 164–185 (1996).

[3] Lasota, S.: Coalgebra morphisms subsume open maps. Theoretical Computer Science, vol. 280, 2002, 123–135.

[4] Pratt, V.R.: Modeling concurrency with geometry. In: 18th Ann. ACM Symp. POPL, pp. 311–322. ACM Press, New York (1991).

A Semantic Hierarchy for Erasure Policies

Filippo Del Tedesco, Chalmers
Sebastian Hunt, City University London
David Sands, Chalmers

1 Introduction

Physical erasure is required when sensitive data must be permanently removed from a memory support. Recent work on erasure for SSD-drives [7] suggests the problem is more subtle than it seems, since low-level routines for erasure often ignore data flows from the drive to the RAM.

Other scenarios require a different notion of erasure. For example, when buying tickets online one expects payment details to be erased from the runtime system at the end of the session. Such *logic erasure* is even more difficult to handle than the physical one. On one hand, it requires more complex policies, that deal with several aspects of the problem, in a spirit similar to the various "dimensions" of declassification, [5]. On the other hand, a proper characterization of logic erasure must involve a detailed description of the attacker observational power. In fact, consider a system which receives some data subject to an erasure policy and then XOR them with a one-time pad random key, which is then overwritten. If an attacker can inspect the final state of the system, erasure turns out to be performed only if he is not able to see the key before it is overwritten, otherwise the secret can be recovered.

The contribution of our work is to formalise (Section 3) a hierarchy of increasingly expressive policies for logic erasure. The hierarchy accounts both different *amounts* of erasure and different varieties of *conditional erasure*. The hierarchy is built on an innovative possibilistic information-flow model (Section 2). Our framework is a novelty with respect to other information-flow approaches to erasure (see, e.g. [1]) because it is parameterised by (i) the *subject* of the information flow policy (e.g. the data to be erased), (ii) the attacker's observational power. In particular, attacker observations and deductions are taken into account by considering the *facts* that an attacker might be interested to learn, and the *queries* which he can or will be able to answer about the subject.

An extended version of this work has been accepted for publication at ICISS2011 [6].

2 An Abstract Model of Information Flow

Systems Representation The behavioural "atoms" in our system representation are *events* (inputs and outputs, for example), arranged in traces. The universe of all possible traces is the set T. A *system* is represented by the set $S \subseteq T$ of its maximal traces.

The portion of a trace which is relevant for erasure, called the *subject* of our policies, is extracted by a projection $\Phi : T \to D$ for some set D. Given a system S, we denote by $\Phi(S)$ the subset of D involved in S, $\Phi(S) = \{\Phi(t) | t \in S\}$. We call this the *subject domain* of S. Let $\mathrm{Sys}(V)$ be the set of all systems with subject domain V. Our flow policies will be specific to systems with a given subject domain.

Visibility Policies The essential component of a flow policy is a *visibility* policy, which specifies how much an attacker should be allowed to learn about the subject of a system by observing its behaviour. Visibility policies are given as equivalence relations, as in many other approaches [2, 4]. The intention is that attackers should not be able to distinguish between subjects which are equivalent according to R.

Some notational conventions will be used in what follows. The set of equivalence relation over V is $ER(V)$, and together with set inclusion it forms a lattice $\langle ER(V), \subseteq \rangle$. The set of equivalence classes of $R \in ER(V)$ is denoted as $[R]$. Since equivalence classes of R partition V in a unique way, R can be expressed as P, a partition of V, such that $P = [R]$. The set of partitions over V is denoted as $PT(V)$ and the lattice $\langle PT(V), \preceq \rangle$ can be derived from $\langle ER(V), \subseteq \rangle$ via isomorphism.

On the Attacker Model Our notion of (passive) attacker is modeled as an equivalence relation on traces, $A \in \mathrm{ER}(T)$. Each equivalence class of A represents system behaviors that look the same from the attacker point of view.

The amount of information about the erasure subject the attacker can deduce from one observation $O \in [A]$ is defined as the *knowledge set* $\mathrm{K}_S(O) = \{\Phi(t) | t \in O \cap S\} \subseteq V$. The union of all possible knowledge sets is called *K-space of A for S*, and it is defined as $\mathscr{K}_S(A) = \{\mathrm{K}_S(O) | O \in [A], O \cap S \neq \emptyset\}$.

Comparing K-spaces and Erasure Policies In general, an erasure policy is satisfied when knowledge sets in the attacker K-space are coarser than the visibility part of the policy. When the K-space is a partition this idea can be easily formalized in terms of refinement: $S \vdash_A R$ iff $[R] \preceq \mathscr{K}_S(A)$, namely S *satisfies R for attacker A* when $\forall D \in [R].v_1, v_2 \in D \rightarrow \exists K \in \mathscr{K}_S(A).v_1, v_2 \in \mathscr{K}_S(A)$.

However, when system's behaviour depends on events which are neither part of the policy subject nor visible to the attacker, the K-space might not be a partition. In this case, a variety of orders are possible, induced by different ways of interpreting the attacker learning strategy.

Facts A fact F is a set of values. A knowledge set X *confirms* F when $X \subseteq F$. Dually, X *has uncertainty* F when $F \subseteq X$. From this we say a K-space K *can confirm* F if there exists some $X \in K$ such that X confirms F, and *can have uncertainty* F if there exists some $X \in K$ such that X has uncertainty F.

Queries A query Q is also a set of values. We say that a given knowledge set X *answers* Q just when either $X \subseteq Q$ or $X \subseteq V \setminus Q$. For a given K-space K we then say that K *will answer* Q if for all $X \in K$, X answers Q, and K *can answer* Q if there exists some $X \in K$ such that X answers Q.

Using the ability of a K-space to confirm facts and answer queries, we can define the following pre-order relations, where a "smaller" K-space allows more deductions.

Order name	Id	Definition
Upper	U	$K_1 \preceq_{\mathrm{U}} K_2$ iff $\forall F.K_2$ can confirm $F \Rightarrow K_1$ can confirm F
Lower	L	$K_1 \preceq_{\mathrm{L}} K_2$ iff $\forall F.K_1$ can have uncertainty $F \Rightarrow K_2$ can have uncertainty F
Convex (Egli-Milner)	EM	$K_1 \preceq_{\mathrm{EM}} K_2$ iff $K_1 \preceq_{\mathrm{U}} K_2 \wedge K_1 \preceq_{\mathrm{L}} K_2$
Can-Answer	CA	$K_1 \preceq_{\mathrm{CA}} K_2$ iff $\forall Q.K_2$ can answer $Q \Rightarrow K_1$ can answer Q
Will-Answer	WA	$K_1 \preceq_{\mathrm{WA}} K_2$ iff $\forall Q.K_2$ will answer $Q \Rightarrow K_1$ will answer Q

Upper and lower orders correspond to the equally named orders in the powerdomain orderings [3].

Orderings have different discrimination power. When a visibility policy is compared with a K-space, it holds that $\preceq_{\mathrm{EM}} \subsetneq \preceq_{\mathrm{L}} \subsetneq \preceq_{\mathrm{WA}}$ and $\preceq_{\mathrm{EM}} \subsetneq \preceq_{\mathrm{U}} \subsetneq \preceq_{\mathrm{CA}}$ and $P \preceq_{\mathrm{CA}} K \Rightarrow P \preceq_{\mathrm{WA}} K$.

3 The policy hierarchy

We now specify our three-levels hierarchy of erasure policy types. A key design principle is that, whenever a policy permits part of the erasure subject to be retained, this should be *explicit*, by which we mean that it should be captured by the conjunction of the component equivalence relations.

Assume a fixed policy subject function $\Phi : T \rightarrow D$. Given a subset $V \subseteq D$, let $T_V = \{t \in T | \Phi(t) \in V\}$. Note that if S belongs to $\mathrm{Sys}(V)$, then $S \subseteq T_V$.

Type 0 policies Type 0 policies allow us to specify *unconditional* erasure, therefore they are just visibility policies. We write Type-0(V) for the set of all Type 0 policies for systems in $\mathrm{Sys}(V)$ (thus Type-0$(V) = \mathrm{ER}(V)$). The definition of satisfaction for a given attacker model A and system S uses a K-space ordering (specified by parameter o) to generalise the satisfaction relation defined in Section 2: $S \vdash^o_A R$ iff $[R] \preceq_{\mathrm{o}} \mathscr{K}_S(A)$.

Type 1 policies Type 1 policies allow us to specify "low dependent" erasure, where different amounts may be erased on different runs, but the erasure condition is independent of the erasure subject itself.

For systems in $\mathrm{Sys}(V)$ the erasure condition is specified as a partition $P \in \mathrm{PT}(T_V)$. This is paired with a function $f : P \rightarrow$ Type-0(V), which associates a Type 0 policy with each element of the partition.

Since the domain of f is determined by the choice of P, we use a dependent type notation to specify the set of all Type 1 policies: Type-1$(V) = \langle P : \mathrm{PT}(T_V), P \to \mathrm{ER}(V)\rangle$.

Because we want to allow only low dependency - the erasure condition must be independent of the erasure subject - we require that P is *total for* V, which means $\forall X \in P.\Phi(X) = V$. This implies knowing the value of the condition will not in itself rule out any possible subject values.

To define policy satisfaction we use the components $X \in P$ to partition a system S into disjoint sub-systems $S \cap X$ and check both that each sub-system is defined over the whole subject domain V (again, to ensure low dependency) and that it satisfies the Type 0 policy for sub-domain X. So, for a Type 1 policy $\langle P, f\rangle \in$ Type-1(V), an attacker model A, and system $S \in \mathrm{Sys}(V)$, satisfaction is defined as $S \vdash^o_A \langle P, f\rangle$ iff $\forall X \in P.S_X \in \mathrm{Sys}(V) \wedge S_X \vdash^o_A f\ X$ where $S_X = S \cap X$.

The following theorem shows our "explicitness" design principle is realised by Type 1 policies:

Theorem 1 Let $\langle P, f\rangle \in$ Type-1(V) and $S \in \mathrm{Sys}(V)$ and $A \in \mathrm{ER}(T)$. Let $o \in \{U, L, EM, CA, WA\}$. If $S \vdash^o_A \langle P, f\rangle$ then: $[\bigwedge_{X \in P}(f\ X)] \preceq_{\mathrm{o}} \mathscr{K}_S(A)$, where $\bigwedge$ is the meet operator in $\langle ER(V), \subseteq\rangle$.

Type 2 policies Type 2 policies are defined as Type-2$(V) = \langle Q : \mathrm{PT}(V), W : Q \to$ Type-1$(W)\rangle$. They allow dependency on both the erasure subject (first component) and other properties of a run (second component).

To define satisfaction for Type 2 policies, we use the components $W \in Q$ to partition a system S into sub-systems (unlike the analogous situation with Type 1 policies, we cannot intersect S directly with W; instead, we intersect with T_W). To ensure that the *only* dependency on the erasure subject is that described by Q, we require that each sub-system $S \cap T_W$ is defined over the whole of the subject sub-domain W. So, for a Type 2 policy $\langle Q, g\rangle \in$ Type-2(V), an attacker model A, and system $S \in \mathrm{Sys}(V)$, satisfaction is defined thus $S \vdash^o_A \langle Q, g\rangle$ iff $\forall W \in Q.S_W \in \mathrm{Sys}(W) \wedge S_W \vdash^o_A g\ W$ where $S_W = S \cap T_W$.

To state the appropriate analogue of Theorem 1 we need to form a conjunction of all the component parts of a Type 2 policy. In the worst case, the attacker will be able to observe which of the erasure cases specified by Q contains the subject. Hence, we should conjoin the corresponding equivalence relation $\mathscr{E}(Q)$. Moreover, each Type 1 sub-policy determines a worst case equivalence relation, as defined in Theorem 1. To conjoin these relations, we must first extend each one from its sub-domain to the whole domain by appending a single additional equivalence class comprising all the "missing" elements: given $W \subseteq V$ and $R \in \mathrm{ER}(W)$, define $R^\dagger \in \mathrm{ER}(V)$ by $R^\dagger = R \cup \bar{W} \times \bar{W}$, where $\bar{W} = V \setminus W$.

Theorem 2 Let $\langle Q, g\rangle \in$ Type-2(V) and $S \in \mathrm{Sys}(V)$ and $A \in \mathrm{ER}(T)$. For any Type 1 policy $\langle P, f\rangle$, let $R_{\langle P,f\rangle} = \bigwedge_{X \in P}(f\ X)$. Let $o \in \{U, L, EM, CA, WA\}$. If $S \vdash^o_A \langle Q, g\rangle$ then $[\ \mathscr{E}(Q) \wedge \bigwedge_{W \in Q} R^\dagger_{(g\ W)}] \preceq_{\mathrm{o}} \mathscr{K}_S(A)$.

References

[1] S. Chong and A.C. Myers. Language-based information erasure. *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 241–254, June 2005.

[2] J. Landauer and T. Redmond. A lattice of information. In *Proc. IEEE Computer Security Foundations Workshop*, pages 65–70, June 1993.

[3] Gordon D. Plotkin. A powerdomain construction. *SIAM J. Comput.*, pages 452–487, 1976.

[4] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1):59–91, March 2001.

[5] A. Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 15(5):517–548, 2009.

[6] F. Del Tedesco, S. Hunt, and D. Sands. A semantic hierarchy for erasure policies. `http://arxiv.org/abs/1109.6914`. The 7th International Conference on Information System Security, Dec 2011.

[7] Michael Yung Chung Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, pages 105–117. USENIX, 2011.

Unifying Synchronous Data and Control Flow in the Lazy λ-Calculus

Joaquín Aguado and Michael Mendler
University of Bamberg, Germany
{joaquin.aguado,michael.mendler}@uni-bamberg.de

Marc Pouzet
ENS Paris, France
Marc.Pouzet@ens.fr

A *Kahn Process Network* (KPN) [9] consists of a set of sequential processes that are connected to one another through uni-directional, buffered data lines for inter-process communications. Each process follows a sequential algorithm. At any given time, it is either computing or waiting for information on one of its input lines (blocking-read). When it receives the required data it may continue to block for more input or compute output. Such output is passed to other processes (in the data-flow path) by placing the results in some of its buffered output lines (non-blocking-write). The individual process nodes of a KPN can be executed concurrently without losing determinism. Kahn's theory [9] provides a compositional model of distributed processes that reconciles the denotational and operational viewpoint. On the other hand, KPN in general depend on global scheduling regimes for proper initialisation and synchronisation to avoid deadlock and buffer overflow.

The *synchronous programming* (SP) technology is one such specialisation of KPN to deal with the scheduling problems. Computations in SP are coordinated under one or more global system clocks (which may be physical or logical) governed by the *synchrony hypothesis* (see e.g., [2]) scheduling scheme. This conveniently abstracts internal, possibly distributed computations into atomic reactions and separates the aspect of determining responsiveness (e.g., clock rate) from the task of verifying functional correctness (state transition function). The hallmark of SP is its ability to control the deterministic execution of interactive processes under static schedules and buffering with well-defined and rigorous Kahn-style semantics. This has resulted in a programming model that is supported by efficient verification, testing and compilation techniques (see, e.g. [4] for a survey on SP) which has led to its success in the design of embedded, reactive and safety-critical systems.

Splitting the declarative and operational viewpoints of Kahn networks, SP technology has produced *data-flow* (DF) languages such as LUSTRE, LUCID SYNCHRONE, SIGNAL or imperative *control-flow* (CF) languages such as STATECHARTS, ESTEREL and QUARTZ. DF takes each communication line as a *stream of data* changing over time and studies the functional and causal relationships between streams. CF packs up all information in the buffers at each point in time and studies the transformations of this global *state* as time progresses. Traditionally, the DF and CF strands of SP have been developed largely independently. Only few attempts have been made to combine the two orthogonal semantics within one language. E.g., [7] adds constructs for hierarchical automata to LUSTRE and defines their semantics by mapping them to data flow clocks, while [6] embeds data flow within parametrised state machines. Those semantics are specialised to particular application contexts and still biased so that control-flow and data-flow are not yet treated in a fully dual and interchangeable fashion as suggested by the KPN model.

In this paper we report on recent work that follows on from [10, 7, 6] tightly to integrate both views in a compositional SP model. We show how the core operators of DF and CF synchronous languages can be combined generically by giving uniform semantics in the lazy λ-calculus. It may thus be implemented as an embedded domain-specific library in any functional programming language, e.g., in Haskell. This yields an abstract view of synchronous KPN that maintains high flexibility for exploiting the trade-offs between DF and CF and which blends naturally with the features of the host language such as higher-order functions, polymorphism, user-defined data types, etc. We will show how this extends previous monadic approaches for imperative streams such as [11], comonadic approaches for data-flow such as [5, 12] or arrows [8]. Our recent work further indicates that synchronous KPN may be an effective

programming pattern to exploit Haskell's light-weight semi-explicit parallelism to speed up functional computations on multi-core machines [1, 3].

Let us briefly sketch the combined DF–CF constructs of our SP model (simplified). The abstract syntax is given by the following BNF with operational semantics as seen in Figure 1 of the appendix:

$$\begin{array}{lll} \text{DF} ::= & rv \mid \texttt{rec}\, rv.\, \text{DF} \mid \texttt{op}\, \text{DF} \ldots \text{DF} \mid \texttt{merge}\, \text{DF}\, \text{DF}\, \text{DF} \mid \text{DF}\, \texttt{when}\, \text{DF} \mid \text{DF}\, \texttt{fby}\, \text{DF} \mid \texttt{srun}\, \text{CF} \\ \text{CF} ::= & s \mid \texttt{loop}\, s.\, \text{CF} \mid sv?x;\, \text{CF} \mid sv!\, \text{DF};\, \text{CF} \mid \texttt{pause};\, \text{CF} \mid \texttt{if}\, \text{DF}\, \texttt{then}\, \text{CF}\, \texttt{else}\, \text{CF} \mid sv := \text{DF} \mid \\ & \texttt{do}\, s = \text{CF}\, \texttt{until}\, sv\, \texttt{then}\, \text{CF} \mid \text{CF}\, \texttt{|>|}\, \text{CF} \mid \texttt{local}\, sv\, \texttt{in}\, \text{CF} \end{array}$$

Each DF represents a stream of values, so the semantics of DF is given most conveniently in terms of a reaction relation $\text{DF} \xrightarrow{v} \text{DF}'$ stating that DF generates a value v and continues as DF′. Streams are recursive and the DF construct `rec` rv`.` DF creates a *feed-back* from the output of DF back to the input *flow variable* rv. The construct binds the variable rv in DF. For simplicity we assume that all DF are closed. The static *lifting* of value operators $\texttt{op}\, \text{DF}_1 \ldots \text{DF}_n$ embeds a standard function `op` canonically, i.e., by element-wise repetition, into streams. A special case are 0-ary operators or constants k which lift to the corresponding infinite stream of constant values. The *up-sampling* $\texttt{merge}\, \text{DF}_b\, \text{DF}_t\, \text{DF}_f$ combines DF_t and DF_f depending on the Boolean stream DF_b. Specifically, if the current value at DF_b is true (`T`) the value sampled is that at DF_t, otherwise if the value at DF_b is false (`F`) the value output is that currently at DF_f. This construct allows slow processes to communicate with faster ones by merging sub-streams into larger DF. The *down-sampling* $\text{DF}\, \texttt{when}\, \text{DF}_b$ extracts sub-streams from DF according to the conditions imposed by Boolean stream DF_b. Concretely, if the value at DF_b is `T` the current value at DF is delivered as output, otherwise, the computation consumes inputs from both DF and DF_b. This allows fast processes to communicate with slower ones. The *unit delay* or *unit buffer* $\text{DF}_1\, \texttt{fby}\, \text{DF}_2$ (read "*followed by*") computes a DF in which the first value is the same as that of DF_1 and the rest is exactly DF_2. Finally, the operator $\texttt{srun}\, \text{CF}$ turns a CF into a DF by recursively closing all instantaneous communications on the signal interface of CF.

Each CF is described by an interface reaction $R \vdash \text{CF} \xrightarrow{R'} \text{CF}'$ that takes an environment stimulus R on its signal interface and produces an instantaneous response R' together with a successor state for the next synchronous instant. A signal environment is a finite mapping $R = [v_1/sv_1, v_2/sv_2 \ldots, v_n/sv_k]$ that associates with every *signal variable* sv_i of the interface a corresponding data value v_i. Write $R[sv_i]$ for the value v_i in R and $R[v/sv]$ for the updating of sv_i with v.

We can create a *state iteration* through the loop statement $\texttt{loop}\, s.\, \text{CF}$. It reacts as CF but with the *state variable* s instantiated by $\texttt{loop}\, s.\, \text{CF}$ itself so that when s is called in the execution of CF the whole process is restarted. To capture explicit memory, CF expressions and state variables s may be parametrised (see e.g. [6]) leading to a general loop construct $\texttt{loop}\, s(x).\, \text{CF}$ where CF depends on a memory parameter x of some type. Here, we only treat the non-parametrised loop and also assume that all state variables appearing in a CF are bound by iteration or weak preemption (see below). This keep the evaluation contexts as simple as possible. The *signal input prefix* $sv?x;\, \text{CF}$ indicates "read the current value v of the signal sv from the environment and then continue as $\text{CF}\{v/x\}$ where *value variable* x is instantiated with value v in CF. Again, for convenience we assume that all value variables in a CF are closed by input prefixes in this way. The *signal emission prefix* $sv!\, \text{DF};\, \text{CF}$ codifies "emit the current value of DF to signal sv in the environment and continue with CF. The *pause* operation $\texttt{pause};\, \text{CF}$ synchronises the CF with the system global clock, waiting for a clock tick before starting CF. It implements the the empty (i.e., identity) reaction in the current instant. The *branching* $\texttt{if}\, \text{DF}\, \texttt{then}\, \text{CF}\, \texttt{else}\, \text{CF}$ operates much in the same fashion as the classical "if-then-else", where depending on the status true (`T`) or false (`F`) derived from DF in the current instant one of the two alternatives CF_1 or CF_2 is taken. The *binding* $sv := \text{DF}$ feeds the values produced by DF to state signal sv instant by instant. The *weak preemption* $\texttt{do}\, s = \text{CF}_1\, \texttt{until}\, sv\, \texttt{then}\, \text{CF}_2$ (written $\texttt{do}\, s = \text{CF}_1\, \texttt{u}\, sv\, \texttt{t}\, \text{CF}_2$ for short) reacts as CF_1 and the next

state depends on a Boolean signal *sv*. If in the current instant the value at *sv* is `T` then the `until` condition is fulfilled and the next state must be the initial state of CF_2 where variable *s* stores the state CF_1' derived from the reaction of CF_1. Otherwise, the `until` condition is not fulfilled and the computation continues in the same way following the states from CF_1. This construct binds the state variable *s* and permits us to build hierarchical state machines with history transitions. *Parallel composition* of states CF_1 `|>|` CF_2 describes parallelism of expressions in a sequential CF fashion to model instantaneous communication. Concretely, CF_1 `|>|` CF_2 first reacts as CF_1 from R producing a response R_1' and then CF_2' reacts from R_1' producing R_2'. The full reaction is the superposition of R_2' in R_1', denoted $R_1' + R_2'$, that is the update of R_1' with all the bindings that occur in R_2' (*i.e.*, the last value dominates). The reaction continues from the parallel composition of the next states of CF_1 and CF_2. The *local signal declaration* `local` *sv* `in` CF declares a lexically scoped signal *sv* to be used for both input and output within CF. This is a recursive binder for signal variables which connects in CF the producers of output values on *sv* with all consumers accessing *sv* as input.

Note that although the rules given in Fig 1 are relational, they have the KPN properties in the sense that they define a deterministic response function for both DF and CF that can be executed in a sequential fashion and thus coded in the (lazy) λ-calculus. We will give more details in the full paper including examples to illustrate the combination of CF and DF within one and the same KPN.

References

[1] J. Aguado and M. Mendler. Computing with streams. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming (DAMP'11)*, pages 35–44. ACM, 2011.

[2] J. Aguado and M. Mendler. Constructive Semantics for Instantaneous Reactions. In *Theoretical Computer Science*, volume 412, pages 931–961, March 2011.

[3] J. Aguado and M. Mendler. An integrated data and control flow programming model. In *Proceedings of INForum, Track Compilers, Programming Languages, Related Technologies and Applications (CoRTA'11)*, pages 234–245, 2011.

[4] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The Synchronous Languages Twelve Years Later. In *Proceedings of the IEEE, Special Issue on Embedded Systems*, volume 91, pages 64–83. IEEE Press, January 2003.

[5] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. Technical Report Research Report 97-7, VERIMAG, Grenoble, October 1997.

[6] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing signals and modes in synchronous data-flow systems. In *ACM Int'l Conf. Embedded Software EMSOFT'06*, 2006.

[7] J.-L. Colaço, B. Pagano, and M. Pouzet. A conservative extension of synchronous data-flow with state machines. In *ACM Int'l Conf. Embedded Software EMSOFT'05*, 2005.

[8] J. Hughes. Programming with arrows. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming (AFP'04)*, volume LNCS 3622, pages 73–129, 2005.

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing: Proceedings of the IFIP Congress '74*, pages 471–475. North-Holland, August 1974.

[10] M. Pouzet. Lucid synchrone, un langage synchrone d'ordre supérieur. Mémoire d'habilitation, Université Paris 6, November 2002.

[11] E. Scholz. Imperative streams - a monadic combinator library for synchronous programming. In *ICFP'98*, pages 261–272, Baltimore, USA, 1998.

[12] T. Uustalu and V. Vene. The essence of dataflow programming. In *CEFP 2005*, pages 135–167. Springer LNCS 4164, 2006.

Appendix I

$$\frac{R[v/sv] \vdash \mathrm{CF} \xrightarrow{R'[v/sv]} \mathrm{CF}'}{\vdash \mathtt{srun}\,\mathrm{CF} \xrightarrow{v} \mathtt{srun}\,\mathrm{CF}'} \qquad \frac{\mathrm{DF}_1 \xrightarrow{v_1} \mathrm{DF}'_1 \ldots \mathrm{DF}_n \xrightarrow{v_n} \mathrm{DF}'_n}{\mathtt{op}\,\mathrm{DF}_1 \ldots \mathrm{DF}_n \xrightarrow{\mathtt{op}\,v_1 \ldots v_n} \mathtt{op}\,\mathrm{DF}'_1 \ldots \mathrm{DF}'_n} \qquad \frac{}{k \xrightarrow{k} k}$$

$$\frac{\mathrm{DF}_b \xrightarrow{\mathtt{T}} \mathrm{DF}'_b \quad \mathrm{DF}_t \xrightarrow{v_t} \mathrm{DF}'_t}{\mathtt{merge}\,\mathrm{DF}_b\,\mathrm{DF}_t\,\mathrm{DF}_f \xrightarrow{v_t} \mathtt{merge}\,\mathrm{DF}'_b\,\mathrm{DF}'_t\,\mathrm{DF}_f} \qquad \frac{\mathrm{DF}_b \xrightarrow{\mathtt{F}} \mathrm{DF}'_b \quad \mathrm{DF}_f \xrightarrow{v_f} \mathrm{DF}'_f}{\mathtt{merge}\,\mathrm{DF}_b\,\mathrm{DF}_t\,\mathrm{DF}_f \xrightarrow{v_f} \mathtt{merge}\,\mathrm{DF}'_b\,\mathrm{DF}_t\,\mathrm{DF}'_f}$$

$$\frac{\mathrm{DF}_b \xrightarrow{\mathtt{T}} \mathrm{DF}'_b \quad \mathrm{DF} \xrightarrow{v} \mathrm{DF}'}{\mathrm{DF}\,\mathtt{when}\,\mathrm{DF}_b \xrightarrow{v} \mathrm{DF}'_b\,\mathtt{when}\,\mathrm{DF}'} \qquad \frac{\mathrm{DF}_b \xrightarrow{\mathtt{F}} \mathrm{DF}'_b \quad \mathrm{DF} \xrightarrow{u} \mathrm{DF}' \quad \mathrm{DF}'\,\mathtt{when}\,\mathrm{DF}'_b \xrightarrow{v} \mathrm{DF}''}{\mathrm{DF}\,\mathtt{when}\,\mathrm{DF}_b \xrightarrow{v} \mathrm{DF}''}$$

$$\frac{\mathrm{DF}_1 \xrightarrow{v} \mathrm{DF}'_1}{\mathrm{DF}_1\,\mathtt{fby}\,\mathrm{DF}_2 \xrightarrow{v} \mathrm{DF}_2} \qquad \frac{\mathrm{DF}\{\mathtt{rec}\,rv.\,\mathrm{DF}/rv\} \xrightarrow{v} \mathrm{DF}'}{\mathtt{rec}\,rv.\,\mathrm{DF} \xrightarrow{v} \mathrm{DF}'}$$

$$\frac{R[sv] = v \quad R \vdash \mathrm{CF}\{v/x\} \xrightarrow{R'} \mathrm{CF}'}{R \vdash sv?x;\,\mathrm{CF} \xrightarrow{R'} \mathrm{CF}'} \qquad \frac{\mathrm{DF} \xrightarrow{v} \mathrm{DF}' \quad R[v/sv] \vdash \mathrm{CF} \xrightarrow{R'} \mathrm{CF}'}{R \vdash sv!\,\mathrm{DF};\,\mathrm{CF} \xrightarrow{R'} \mathrm{CF}'}$$

$$\frac{}{R \vdash \mathtt{pause};\,\mathrm{CF} \xrightarrow{R} \mathrm{CF}} \qquad \frac{\mathrm{DF} \xrightarrow{v} \mathrm{DF}'}{R \vdash sv := \mathrm{DF} \xrightarrow{R[v/sv]} sv := \mathrm{DF}'} \qquad \frac{R \vdash \mathrm{CF}\{(\mathtt{loop}s.\,\mathrm{CF})/s\} \xrightarrow{R'} \mathrm{CF}'}{R \vdash \mathtt{loop}s.\,\mathrm{CF} \xrightarrow{R'} \mathrm{CF}'}$$

$$\frac{\mathrm{DF}_b \xrightarrow{\mathtt{T}} \mathrm{DF}'_b \quad R \vdash \mathrm{CF}_1 \xrightarrow{R'} \mathrm{CF}'_1}{R \vdash \mathtt{if}\,\mathrm{DF}_b\,\mathtt{then}\,\mathrm{CF}_1\,\mathtt{else}\,\mathrm{CF}_2 \xrightarrow{R'} \mathrm{CF}'_1} \qquad \frac{\mathrm{DF}_b \xrightarrow{\mathtt{F}} \mathrm{DF}'_b \quad R \vdash \mathrm{CF}_2 \xrightarrow{R'} \mathrm{CF}'_2}{R \vdash \mathtt{if}\,\mathrm{DF}_b\,\mathtt{then}\,\mathrm{CF}_1\,\mathtt{else}\,\mathrm{CF}_2 \xrightarrow{R'} \mathrm{CF}'_2}$$

$$\frac{R[sv] = \mathtt{T} \quad R \vdash \mathrm{CF}_1 \xrightarrow{R'} \mathrm{CF}'_1}{R \vdash \mathtt{do}s\mathtt{=}\mathrm{CF}_1\,\mathtt{u}sv\mathtt{t}\,\mathrm{CF}_2 \xrightarrow{R'} \mathrm{CF}_2\{\mathrm{CF}'_1/s\}} \qquad \frac{R[sv] = \mathtt{F} \quad R \vdash \mathrm{CF}_1 \xrightarrow{R'} \mathrm{CF}'_1}{R \vdash \mathtt{do}s\mathtt{=}\mathrm{CF}_1\,\mathtt{u}sv\mathtt{t}\,\mathrm{CF}_2 \xrightarrow{R'} \mathtt{do}s\mathtt{=}\mathrm{CF}'_1\,\mathtt{u}sv\mathtt{t}\,\mathrm{CF}_2}$$

$$\frac{R \vdash \mathrm{CF}_1 \xrightarrow{R'_1} \mathrm{CF}'_1 \quad R'_1 \vdash \mathrm{CF}_2 \xrightarrow{R'_2} \mathrm{CF}'_2}{R \vdash \mathrm{CF}_1 \mid\!> \mid \mathrm{CF}_2 \xrightarrow{R'_1+R'_2} \mathrm{CF}'_1 \mid\!> \mid \mathrm{CF}'_2} \qquad \frac{R[v/sv] \vdash \mathrm{CF} \xrightarrow{R'[v/sv]} \mathrm{CF}'}{R \vdash \mathtt{local}\,sv\,\mathtt{in}\,\mathrm{CF} \xrightarrow{R'} \mathtt{local}\,sv\,\mathtt{in}\,\mathrm{CF}'}$$

Figure 1: Operational Semantics of Simple CF-DF language

Inheritance and Observability

Erika Ábrahám, Thi Mai Thuong Tran, and Martin Steffen

RWTH Aachen, Germany and University of Oslo, Norway

An *open* system is a part of a larger system, which interacts with its environment, and best considered as a black box where the internals are hidden. Such a separation of internal behavior from externally relevant interface behavior is crucial for compositionality. The most popular programming paradigm nowadays is object orientation, which in particular supports interfaces and encapsulation of objects. Another crucial feature in mainstream object orientation is *inheritance,* which allows code reuse and is intended to support incremental program development by gradually extending and specializing an existing class hierarchy.

Openness of a system in the presence of inheritance and late binding is problematic. One symptom of that is known in software engineering as the fragile base class problem. A base class in an inheritance hierarchy is a (common) super-class, and fragile means that replacing one base class by another, seemingly satisfying the same interface description, may break the code of the client of the base class, i.e., change the behavior of the "environment" of the base class.

A rigorous method to keep track of interactional behaviours of an open program is the key to formal verification of open programs as well as a formal foundation for black-box testing. If done properly, it ultimately allows compositional reasoning, i.e., to infer properties of a composed system from the interface properties of its sub-constituents without referring to further internal representation details. A representation-independent, abstract account of the behavior is also necessary for compositional optimization of components: only when showing the same external behavior one program can replace another without changing the interaction with any client code.

An object-oriented, concurrent calculus The calculus presented in this work is a concurrent variant of an imperative, object-calculus. Its concurrency model is based on the notion of "active objects" and asynchronous method calls. Its syntax is sketched in Table 1 focusing on important features, such as: classes with fields and methods, objects as instances of classes, and concurrency based on the active objects model of concurrency. Expressions e basically consists of a sequential composition (here represented by the let-construct) of basic expressions, including conditionals, object creation, read and write of object fields, and method calls.

Being standard, the syntax should be largely clear, a few points are worth highlighting though: the concurrency model of the calculus based on *active objects,* communicating via *asynchronous* method calls (written $o@l(\boldsymbol{v})$) and the result is given back by the caller querying a future reference. Objects act as monitors with binary locks. The notion of single inheritance based on classes, however, is orthogonal to the choice of the concurrency model.

$C ::= \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[\![O]\!] \mid \underline{n[O, L]} \mid \underline{n\langle t\rangle}$		component
$O ::= n, M, F$		object
$M ::= l = m, \ldots, l = m$		method suite
$F ::= l = f, \ldots, l = f$		fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$		method
$f ::= v \mid \bot_{n'}$		field
$t ::= v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e\, \mathsf{in}\, t$		thread
$e ::= t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}(v.l())\, \mathsf{then}\, e\, \mathsf{else}\, e$		expr.
$\mid\ n@l(\boldsymbol{v}) \mid v.l() \mid v.l() := v$		
$\mid\ \mathsf{new}\, n \mid \mathsf{claim@}(n, n) \mid \underline{\mathsf{get@}n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$		
$v ::= x \mid n \mid ()$		values
$L ::= \bot \mid \top$		lock status

Table 1. Syntax of an oo core calculus

Typed operational semantics for interface behavior In this setting, the component behavior consists of message traces, i.e., sequences of component-environment interactions. Writing $C \stackrel{t}{\Longrightarrow} \acute{C}$, the t denotes the *trace* of interface actions by which C evolves into $\acute{C}$, potentially executing internal steps, as well, not recorded in t. An open program C, however, does not act in isolation, but interacts with *some* environment. I.e., we are interested in traces t where *there exists an environment* E such that $C \parallel E \stackrel{t}{\underset{\bar{t}}{\Longrightarrow}} \acute{C} \parallel \acute{E}$ by which we mean: component C produces the trace t and E produces the dual trace $\bar{t}$, both together "canceling out" to internal steps. In other words, our goal is to formulate the external or open semantics with the environment *existentially abstracted away.* With infinitely many possible environments E, the challenge is to capture what is common to *all* those environments. This will be done in form of *assumptions* about the environment. This means, the operational semantics specifies the behavior of C under certain assumptions Ξ_E about the environment. Following standard notation from logics, we do not write $\Xi_E \parallel C$, but rather $\Xi_E \vdash C$, such that the reductions will look like

$$\Xi_E \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi}_E \vdash \acute{C}\ . \tag{1}$$

Such a characterization of the abstract interface behavior is relevant for the following reasons. Firstly: the set of traces according to equation (1) is more restricted than the one obtained when ignoring the environments altogether. This means, when *reasoning* about the behavior of C based on the traces, e.g., for the purpose of verification, the more precise knowledge of the possible traces allows to carry out stronger arguments about C. Secondly, an application for a trace description is black-box testing, in that one describes the behavior of a component in terms of the interface traces and then synthesize appropriate test drivers from it. Obviously it makes no sense to specify interface behavior which

is not possible, at all, since in this case one could not generate a corresponding tester. Finally, and not as the least gain, the formulation gives *insight* into the inherent semantical nature of the language, as the assumptions Ξ and the semantics captures the existentially abstracted environment behavior.

Main results

- A formal, open semantics for a statically typed, concurrent object-oriented calculus with dynamic object creation, mutable heap, and single *inheritance.*
- The main insight of our work is that the cross-border inheritance complicates the observable behavior considerably. Namely, in an open setting, environment and component classes can inherit from each other. Therefore an object may contain fields defined by the components and by the environment. Due to privacy restrictions, these fields can only be manipulated by the corresponding methods of environment resp. component parts. To describe the possible interface behavior, where all possible environments are existentially abstracted away and represented by an assumption context, the potential connectivity of the enviroment part of the heap is important. In order to capture that, our open semantics must be able to tell when a communication between two objects is possible, i.e, when they are potentially in connnection.
- The interface behavior is characterized in the form of a typed operational semantics of an open system, consisting of a set of classes.
- The semantics is formalized in the form of commitments of the component and in particular *assumptions* about the environment. The fact that the components are open wrt. inheritance, i.e., a component can inherit from the environment and vice versa, has as a consequence that the assumptions and commitments need contain an abstraction of the heap topology, keeping track of which object may be in connection with other objects.
- Finally, we show the soundness of the abstractions based on trace semantics.

More details can be found in [1]

References

1. E. Ábrahám, T. Mai Thuong Tran, and M. Steffen. Observable interface behavior and inheritance. Technical Report 409, University of Oslo, Dept. of Informatics, Apr. 2011. `www.ifi.uio.no/~msteffen/publications.html#techreports`.

Compositional Transfinite Semantics of **While**

Härmel Nestra*
Institute of Computer Science, University of Tartu
harmel.nestra@ut.ee

In *transfinite semantics*, program executions are conceived as continuing after infinite loops from some limit states. Transfinite semantics provides a way to overcome so-called semantic anomaly in program transformation such as slicing.

Roughly, *program slicing* [13, 2, 12] means leaving out parts of the program that do not influence values of some variables at some fixed program points. The result must be an executable program which, when run with the same initial state as the original program, brings forth the same sequence of values of the interesting variables at the interesting program points. For instance, if we are interested in only the value of variable `sum` at the end of the execution of the following folklore program, the two lines that assign something to `prod` can be left out:

```
n := input() ;
i := 0 ;
sum := 0 ;
prod := 1 ;
while i < n do
(
  i := i + 1 ;
  sum := sum + i ;
  prod := prod * i
)
```

In particular, dead code can always be sliced away. The classic slicing algorithms use control and data flow analysis for determining parts that cannot influence the computation of the interesting values. This analysis marks loops, influence of which to the interesting variables via control and data flow is not recognized, as entirely irrelevant. However, if such a loop does not terminate then the execution of the resulting program where this loop has been removed can reach farther in the code than the original program and assign values to interesting variables that are never assigned during the run of the original program. This phenomenon is called *semantic anomaly* [6].

Semantic anomaly is closely connected to the feature of standard semantics that executions can make at most ω (the least infinite ordinal number) steps. Transfinite semantics where any infinite execution continues after the body of a loop has been executed ω times is able to overcome this discrepancy. This approach has been proposed by [3, 4] and investigated further by Giacobazzi and Mastroeni [6] and by us [7, 9, 8, 10] (some other approaches are described in [5, 1]). In order to match the flow analysis based algorithms, the limit state from which the computation goes on after an infinite loop must satisfy the following property: if the value of any variable observed at the top point of the loop is permanently v starting from some stage of the execution then this variable has value v also in the limit state [7].

Transfinite semantics have been criticized because in the forms they have been proposed, they have often had rather artificial construction and lacked of desired properties such as compositionality and substitutivity. (The transfinite semantics of [6] is sequential rather than compositional.) Barraclough et al. [1] recently proposed another approach where traces are replaced with sequences of traces. In the nth component of a trace sequence, the number of iterations of each loop body during one execution of the

*This work is partially supported by Estonian Science Foundation under grants no. 7543 and 8421

loop is bounded by n (after that, the execution is continued from the program point following the loop with the state reached). Note that all traces in this semantics are finite (even in the case of infinite loops). This semantics, called *trajectory semantics*, is proven to be substitutive by [1].

A disadvantage of the semantics of [1] is that deciphering the standard semantics from it is not straightforward. This can be done via "diagonalization" (the nth state in the standard semantics trace is the nth state on the nth trace in the trajectory semantics) but this characterization is probably hard to use in practice. On the other hand, obtaining standard semantics from transfinite semantics could be done by just truncating the transfinite parts of all infinite traces (standard semantics is always included in the transfinite semantics).

In our work in progress, we construct a compositional (thus also substitutive) transfinite semantics for **While** suitable for use in program slicing theory. We show that standard trace semantics is its straightforward abstraction in the sense just described. We also establish its connections to transfinite semantics given in the greatest fixpoint form that matches another standard way of representing semantics.

As shown earlier [10], the usual trace semantics where intermediate states are indexed by ordinal numbers is not suitable for expressing transfinite semantics in the form of greatest fixpoint. For this purpose, fractional semantics or tree semantics must be used. In tree semantics, traces of executions are replaced with trees that reverberate the structure of deduction of the assertion that the execution is valid in the semantics. This is close to natural semantics [11] but can have some peculiarities (like trees with one or more infinite branches). In fractional semantics [8, 10], trace components are indexed by rational numbers between 0 and 1. This way, it is possible to capture both the concept of linearly progressing trace and the deduction tree structure. To each statement of the program, fixed intervals of rational numbers are associated according to the structure of the program.

For example, the fractional execution traces of programs `(z := x ; x := y) ; y := z` and `z := x ; (x := y ; y := z)` in the initial state $\left\{\begin{array}{l}\mathtt{x}\mapsto 1\\ \mathtt{y}\mapsto 2\\ \mathtt{z}\mapsto 0\end{array}\right\}$ are

$$\left(0\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 1\\ \mathtt{y}\mapsto 2\\ \mathtt{z}\mapsto 0\end{array}\right\},\frac{1}{4}\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 1\\ \mathtt{y}\mapsto 2\\ \mathtt{z}\mapsto 1\end{array}\right\},\frac{1}{2}\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 2\\ \mathtt{y}\mapsto 2\\ \mathtt{z}\mapsto 1\end{array}\right\},1\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 2\\ \mathtt{y}\mapsto 1\\ \mathtt{z}\mapsto 1\end{array}\right\}\right),$$

$$\left(0\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 1\\ \mathtt{y}\mapsto 2\\ \mathtt{z}\mapsto 0\end{array}\right\},\frac{1}{2}\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 1\\ \mathtt{y}\mapsto 2\\ \mathtt{z}\mapsto 1\end{array}\right\},\frac{3}{4}\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 2\\ \mathtt{y}\mapsto 2\\ \mathtt{z}\mapsto 1\end{array}\right\},1\mapsto\left\{\begin{array}{l}\mathtt{x}\mapsto 2\\ \mathtt{y}\mapsto 1\\ \mathtt{z}\mapsto 1\end{array}\right\}\right),$$

respectively. Note that, in each case, composition divides the index space into two equal halves.

In this work, we use fractional semantics since the linear shape of traces tends to simplify proofs.

Another issue is raised by the restriction imposed on the limit states observed above. In order to implement the restriction into the semantics, the states observed at the top point of a loop must be somehow distinguished. This was discussed in [10] already though the semantics studied there did not take the restriction into account (and thus included too many traces). That discussion suggested explicit tracing of program points in semantics. It turns out that one can do it simpler. In the compositional construction of semantics, one can just take the initial state of each iteration of the loop body, and in the greatest fixpoint form, the fractional shape of traces enables deciphering the program points from the indices.

So we have two classifications of semantics involved: standard vs transfinite semantics, and ordinal vs fractional semantics. This is 4 semantics altogether, that we define uniformly using a parametric framework like in [10]. In the detailed study, we concentrate on 3 semantics since using fractional shape of traces is not reasonable in the case of standard semantics. Unlike in [10], all semantics are explicitly compositional. To summarize the concrete results:

- The semantics involving the largest amount of information, i.e. the transfinite fractional semantics, is expressible in the form of greatest fixpoint.
- The classic slicing algorithms work correctly w.r.t. the transfinite fractional semantics.
- The first abstraction: the transfinite ordinal semantics can be obtained from the fractional one by renumbering the states by ordinals while keeping the order.
- The second abstraction: the standard ordinal semantics can be obtained from the previous one by cutting off all parts of traces that go beyond the first ω steps.

The proof of the first abstraction step is more or less straightforward but that of the second one (abandoning transfiniteness) is surprisingly hard although the abstraction function (truncating the transfinite parts of traces) is simple and intuitive. The difficulties arise in the case of iteration: executing the body of a loop may itself be either finite or infinite, therefore the stage that introduces the first infinity may vary.

As a general consequence, the work confirms that using transfinite semantics to formalize program slicing is still motivated. Another conclusion to be made is that fractional semantics is useful for more reasons than it has been previously stated. Besides defining transfinite semantics for recursive programs [8] and expressing transfinite semantics in the form of greatest fixpoint [10], it may let one avoid explicit program points in semantics descriptions while keeping the necessary information provided by program points available otherwise. This applies also to defining the correspondence of program points of the slice and of the original program. If the redundant statements are replaced with **`skip`** rather than removed (an equivalent alternative for defining program slicing) then it can be obtained by just comparing the state indices in fractional semantics (the states observed at corresponding program points have equal indices).

References

[1] Barraclough, R. W., Binkley, D., Danicic, S., Harman, M., Hierons, R. M., Kiss, Á., Laurence, M., Ouarbya, L.: A trajectory-based strict semantics for program slicing. *Theoretical Computer Science* **410** (2010) 1372–1386

[2] Binkley, D. W., Gallagher, K. B.: Program slicing. *Advances in Computers* **43** (1996) 1–50

[3] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science* **6** (1997) 25 pp.

[4] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science* **277** (2002) 47–103

[5] Danicic, S., Harman, M., Howroyd, J., Ouarbya, L.: A non-standard semantics for program slicing and dependence analysis. *Journal of Logic and Algebraic Programming* **72** (2007) 191–206

[6] Giacobazzi, R., Mastroeni, I.: Non-standard semantics for program slicing. *Higher-Order Symbolic Computation* **16** (2003) 297–339

[7] Nestra, H.: Transfinite semantics in program slicing. *Proceedings of the Estonian Academy of Sciences: Engineering* **11**(4) (2005) 313–328

[8] Nestra, H.: Fractional semantics. In Johnson, M., Vene, V. (eds.): *Proceedings of AMAST 2006*. *Lecture Notes in Computer Science* **4019** (2006) 278–292

[9] Nestra, H.: Iteratively Defined Transfinite Trace Semantics and Program Slicing with respect to Them. PhD thesis, University of Tartu (2006) 119 pp.

[10] Nestra, H.: Transfinite semantics in the form of greatest fixpoint. *Journal of Logic and Algebraic Programming* **78** (2009) 573–592

[11] Nielson, F, Nielson, H. R.: *Semantics with Applications: An Appetizer*. Springer (2007)

[12] Tip, F.: A survey of program slicing techniques. *Journal of Programming Languages* **3**(3) (1995) 121–181

[13] Weiser, M.: Programmers use slices when debugging. *Communications of the ACM* **25**(7) (1982) 446–452