25th Nordic Workshop on Programming Theory

NWPT 2013

Tallinn, Estonia, 20-22 November 2013

Abstracts

25th Nordic Workshop on Programming Theory

NWPT 2013

Tallinn, Estonia, 20–22 November 2013

Abstracts

Institute of Cybernetics at Tallinn University of Technology

Tallinn ∘ 2013

25th Nordic Workshop on Programming Theory
NWPT 2013
Tallinn, Estonia, 20–22 November 2013
Abstracts

Edited by Tarmo Uustalu and Jüri Vain

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
http://www.ioc.ee/

Department of Computer Science, Tallinn University of Technology
Akadeemia tee 15A, 12618 Tallinn, Estonia
http://cs.ttu.ee/

ISBN 978-9949-430-70-3 (pdf)

Preface

This volume contains the abstracts of the talks to be presented at the 25th Nordic Workshop on Programming Theory, NWPT 2013, to take place in Tallinn, Estonia, 20–22 November 2013.

The NWPT workshops are a forum bringing together programming theorists from the Nordic and Baltic countries (but also elsewhere). The previous workshops were held in Uppsala (1989, 1999 and 2004), Aalborg (1990), Göteborg (1991 and 1995), Bergen (1992, 2000 and 2012), Åbo (Turku) (1993, 1998, 2003 and 2010), Aarhus (1994), Oslo (1996, 2007), Tallinn (1997, 2002 and 2008), Lyngby near Copenhagen (2001 and 2009), Copenhagen (2005), Reykjavík (2006) and Västerås (2011). This year 2013 the workshop series celebrates its 25th anniversary.

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

The programme of NWPT 2013 includes three invited talks by distinguished researchers—Keijo Heljanko (Aalto University), Shin-ya Katsumata (Kyoto University) and Jaco van de Pol (Universiteit Twente). The contributed part of the programme consists of 23 talks by authors from different European countries.

NWPT 2013 is sponsored by the European Regional Development Fund through the Estonian Centre of Excellence in Computer Science, EXCS.

Tarmo Uustalu and Jüri Vain

Tallinn, 15 November 2013

Organization

Programme Committee

Luca Aceto (Reykjavík University)
Lars Birkedal (Aarhus Universitet)
Michael Reichhardt Hansen (Danmarks Tekniske Universitet)
Einar Broch Johnsen (Universitetet i Oslo)
Yngve Lamo (Høgskolen i Bergen)
Kim Guldstrand Larsen (Aalborg Universitet)
Mohammad Reza Mousavi (Högskolan i Halmstad)
Bengt Nordström (Chalmers Tekniska Högskola)
Olaf Owe (Universitetet i Oslo)
Paul Pettersson (Mälardalens Högskola)
Andrei Sabelfeld (Chalmers Tekniska Högskola)
Gerardo Schneider (Chalmers Tekniska Högskola)
Tarmo Uustalu (Institute of Cybernetics at TUT, co-chair)
Jüri Vain (Tallinn University of Technology, co-chair)
Marina Waldén (Åbo Akademi)
Uwe Wolter (Universitetet i Bergen)
Wang Yi (Uppsala Universitet)

Organizing Committee

Juhan Ernits (Tallinn University of Technology)
Alisa Pankova (Cybernetica AS)
Monika Perkmann (Institute of Cybernetics at TUT)
Tarmo Uustalu (Institute of Cybernetics at TUT)

Hosts

Institute of Cybernetics at Tallinn University of Technology
Department of Computer Science, Tallinn University of Technology

Sponsor

European Regional Development Fund
through the Estonian Centre of Excellence in Computer Science, EXCS

Table of Contents

Invited talks

Contributed talks

Using Unfoldings in Automated Testing of Multithreaded Programs

Keijo Heljanko

Dept. of Computer Science and Engineering, Aalto University,
PO Box 15400, 00076 Aalto, Finland
keijo.heljanko@aalto.fi

In multithreaded programs both environment input data and the nondeterministic interleavings of concurrent events can affect the behavior of the program. One approach to systematically explore the nondeterminism caused by input data is dynamic symbolic execution. For testing multithreaded programs we present a new approach that combines dynamic symbolic execution with unfoldings, a method originally developed for Petri nets but also applied to many other models of concurrency. We provide an experimental comparison of our new approach with existing algorithms combining dynamic symbolic execution and partial-order reductions and show that the new algorithm can explore the reachable control states of each thread with a significantly smaller number of test runs. In some cases the reduction to the number of test runs can be even exponential allowing programs with long test executions or hard-to-solve constrains generated by symbolic execution to be tested more efficiently.

(This is joint work with Kari Kähkönen and Olli Saarikivi.)

Relating Computational Effects by $\top\top$-Lifting

Shin-ya Katsumata

Research Institute for Mathematical Sciences (RIMS), Kyoto University,
Kitashirakawa Oiwakecho, Sakyoku, Kyoto 606-8502, Japan
sinya@kurims.kyoto-u.ac.jp

When we have two implementations of a programming language, we are naturally interested in knowing a relationship between these implementations. Suppose that we obtain two relationships on 1) data representations in two implementations and 2) behaviours of side-effects in two implementations. Then the question is whether the obtained relationships are respected by every program, that is, for every program P, when related data are supplied as input, the execution of P in two implementations raises related side-effects.

We consider this question in the following theoretical setting:

- For programming languages, we employ λ_c calculi extended with algebraic operations. We view them as idealised call-by-value functional programming languages.
- For implementations, we employ Moggi's monadic semantics of λ_c-calculi.

We present a sufficient condition for a given set of relationships for values and computations to be respected by every program. This condition is natural, and applicable to any λ_c-calculus with algebraic operations, monadic semantics and relationships under consideration.

The proof of the condition being sufficient hinges on the technique called categorical $\top\top$-lifting. It is a semantic formulation of Lindley and Stark's leapfrog method [3, 4], and constructs logical relations for monads. This construction takes a parameter, and by varying it we can derive various logical relations. In the proof of the sufficiency, we supply the relationship on computations as the parameter—this is the key to achieve the generality of the condition.

If time permits, I will talk about other applications of the categorical $\top\top$-lifting. This talk is based on [1, 2].

References

[1] S. Katsumata. A semantic formulation of $\top\top$-lifting and logical predicates for computational metalanguage. In L. Ong, ed., *Proc. CSL 2005*, *Lect. Notes Comput. Sci.*, v. 3634, pp. 87–102. Springer, 2005.

[2] S. Katsumata. Relating computational effects by $\top\top$-lifting. *Inf. Comput.*, v. 222, pp. 228–246, 2013.

[3] S. Lindley. Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages. PhD thesis, University of Edinburgh, 2004.

[4] S. Lindley, I. Stark. Reducibility and $\top\top$-lifting for computation types. In P. Urzyczyn, ed., *Proc. TLCA 2005*, *Lect. Notes Comput. Sci.*, v. 3461, pp. 262–277. Springer, 2005.

Multi-Core Model Checking for Biological Applications

Jaco van de Pol

Centre for Telematics and Information Technology, Universiteit Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands
vdpol@cs.utwente.nl

Multi-core model checking algorithms aim at speeding up verification tasks, by using multiple processor cores running in shared memory. Using shared memory avoids communication overhead due to message passing, but it is far from trivial to obtain ideal speedups, since the underlying graph algorithms are memory intensive and irregular.

I will present some key ideas in multi-core model checking, which have enabled us to provide scalable solutions for reachability, LTL model checking, and symbolic model checking. Key ingredients are a scalable shared hashtable, parallel random search algorithms, and an efficient work stealing scheme to implement multi-core decision diagrams.

We implemented our ideas in the LTSmin toolset, providing a high performance model checker through the PINS interface, independent of the specification language. We instantiated this for Promela, mCRL, DVE and UPPAAL timed automata. I will explain how a slight extension of PINS enables LTL model checking for timed automata.

Productive Infinite Objects via Copatterns

Andreas Abel

Department of Computer Science and Engineering
Chalmers and Gothenburg University
Gothenburg, Sweden
andreas.abel@gu.se

Inductive data such as lists and trees is modeled category-theoretically as *algebra* where *construction* is the primary concept and elimination is obtained by initiality. In a more practical setting, functions are programmed by *pattern matching* on inductive data. Dually, coinductive structures such as streams and processes are modeled as *coalgebras* where *destruction* (or transition) is primary and construction rests on finality [Hag87]. Due to the coincidence of least and greatest fixed-point types [SP82] in lazy languages such as Haskell, the distinction between inductive and coinductive types is blurred in partial functional programming. As a consequence, coinductive structures are treated just as infinitely deep (or, non-well-founded) trees, and pattern matching on coinductive data is the dominant programming style. In total functional programming, which is underlying the dependently-typed proof assistants Coq [INR12] and Agda [Nor07], the distinction between induction and coinduction is vital for the soundness, and pattern matching on coinductive data leads to the loss of subject reduction [Gim96]. Further, in terms of expressive power, the *productivity checker* for definitions by coinduction lacks behind the termination checker for inductively defined functions.

It is thus worth considering the alternative picture that a *coalgebraic approach* to coinductive structures might offer for total and, especially, for dependently-typed programming. The coalgebraic approach as pioneered by Hagino has been followed in the design of the language *Charity* [CF92] and advocated by Setzer for use in Type Theory [Set12]. Now, if "algebraic programming" amounts to defining functions by pattern matching, what is "coalgebraic programming"? Or, asked otherwise, what is the proper dualization of pattern matching, what is *copattern* matching?

While patterns match the introduction forms of finite data, copatterns match on elimination contexts for infinite objects, which are applications (eliminating functions) and destructors/projections (eliminating coalgebraic types = Hagino's codatatypes = Cockett's final datatypes). An infinite object such as a function or a stream can be defined by its behavior in all possible contexts. Thus, if we consider a set of copatterns covering all possible elimination contexts, plus the object's response for each of the copatterns, that object is defined uniquely. More concretely, a stream is determined by its head and its tail, thus, we can introduce a new stream object by giving two equations; one that specifies the value it produces if its head is demanded, and one for the case that the tail is demanded. Another covering set of copatterns consists of head, head of tail, and tail of tail. For instance, the stream of Fibonacci numbers can be given by the three equations, using a function $\mathsf{zipWith}\ f\ s\ t$ which pointwise applies the binary function f to the elements of streams s and t.

$$
\begin{aligned}
\mathsf{zipWith}\ f\ s\ t\ .\mathsf{head} &= f\ (s\ .\mathsf{head})\ (t\ .\mathsf{head}) \\
\mathsf{zipWith}\ f\ s\ t\ .\mathsf{tail} &= \mathsf{zipWith}\ f\ (s\ .\mathsf{tail})\ (t\ .\mathsf{tail}) \\
\\
\mathsf{fib}\ .\mathsf{head} &= 0 \\
\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{head} &= 1 \\
\mathsf{fib}\ .\mathsf{tail}\ .\mathsf{tail} &= \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\ .\mathsf{tail})
\end{aligned}
$$

Taking the above equations as left-to-right rewrite rules, we obtain a strongly normalizing system. This is in contrast to the conventional definition of fib in terms of the stream constructor $h :: t$ by

$$\mathsf{fib} = 0 :: 1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\,.\mathsf{tail})$$

which, even if unfolded under destructors only, admits an infinite reduction sequence starting with $\mathsf{fib}\,.\mathsf{tail} \longrightarrow 1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\,.\mathsf{tail}) \longrightarrow 1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (1 :: \mathsf{zipWith}\ (+)\ \mathsf{fib}\ (\mathsf{fib}\,.\mathsf{tail}))$ $\longrightarrow \ldots$ The crucial difference is that $\mathsf{fib}\,.\mathsf{tail}$ does not reduce if we choose the definition by copatterns above, since the elimination .tail is not matched by any of the copatterns; only in contexts .head or .tail .head or .tail .tail it is that fib springs into action.

Using definitions by copattern matching, we reduce productivity to termination and productivity checking to termination checking. As termination of a function is usually proven by a measure on the size of the function arguments, we prove productivity by well-founded induction on the size of the elimination context. For instance, fib is productive because the recursive calls occur in smaller contexts: at least one tail-destructor is "consumed" and, equally important, zipWith does not add any more destructors. The number of eliminations (as well as the size of arguments) can be tracked by sized types [HPS96], reducing productivity (and termination) checking to type checking. For a polymorphic lambda-calculus with inductive and coinductive types and patterns and copatterns, this has been spelled out in joint work with Brigitte Pientka [AP13]. An introductory study of copatterns and covering sets thereof can be found in previous work [APTS13].

References

[AP13] Andreas Abel and Brigitte Pientka. Wellfounded recursion with copatterns: A unified approach to termination and productivity. In *International Conference on Functional Programming (ICFP 2013)*. ACM Press, 2013.

[APTS13] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: Programming infinite structures by observations. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'13, Rome, Italy, January 23 - 25, 2013*, pages 27–38. ACM Press, 2013.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Technical report, Department of Computer Science, The University of Calgary, 1992. Yellow Series Report No. 92/480/18.

[Gim96] Eduardo Giménez. *Un Calcul de Constructions Infinies et son application a la vérification de systèmes communicants*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996. Thèse d'université.

[Hag87] Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423, 1996.

[INR12] INRIA. *The Coq Proof Assistant Reference Manual*. INRIA, version 8.4 edition, 2012.

[Nor07] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2007.

[Set12] Anton Setzer. Coalgebras as types determined by their elimination rules. In *Epistemology versus Ontology: Essays on the Philosophy and Foundations of Mathematics in Honour of Per Martin-Löf*, volume 27 of *Logic, Epistemology, and the Unity of Science*, pages 351–369. Springer-Verlag, 2012.

[SP82] Michael B. Smyth and Gordon D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11(4):761–783, 1982.

Behavioral Comparison of Acyclic Business Process Models

Abel Armas-Cervantes[1], Paolo Baldan[2] and Luciano García-Bañuelos[1]

[1] Institute of Computer Science, University of Tartu, Estonia
{abel.armas,luciano.garcia}@ut.ee
[2] Department of Mathematics, University of Padova, Italy.
baldan@math.unipd.it

1 Introduction

This work presents an approach for behavioral comparison of acyclic process models – specifically acyclic ones as a starting point. Concretely, given two acyclic process models, we want to determine if they are behaviorally equivalent. If they are not equivalent, we want to highlight their differences using simple statements. For instance, if we consider the process models in Figure 1, we aim at providing the following statement: *"In the first process model (a) tasks a and e never appear in the same run; whereas, in the second process (b) there exists a run where both tasks occur"*. In order to derive such statements, we need to compare the behavior signature of each input process model, expressed in its more basic form: binary behavioral relations (e.g., causality, conflict, concurrency, etc.)

One such representation is given by Behavioral profiles [5]. In this representation, the behavior of a process model is encoded in a $n \times n$ matrix, where n represents the number of tasks in the process. Each cell in the matrix stores the behavior relation observed over the corresponding pair of tasks. However, behavioral profiles have major issues still to be resolved: a) mishandle the cases when there are duplicate tasks, b) they do not correspond to any well-accepted notion of behavioral equivalence – i.e., two models can have the same matrix representation even if their behavior is different–, and c) as a consequence of the above, it fails to diagnose various types of behavioral differences.

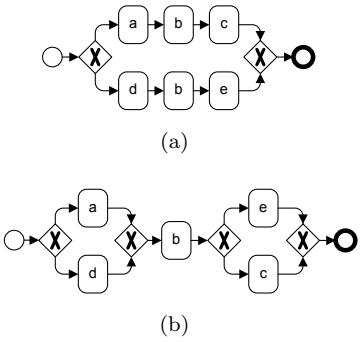


Figure 1: Running example

In light of the above, our research goals are the following. First, we aim at producing a representation of process behavior based on binary relations and producing accurate difference diagnostics under a well-accepted notion of equivalence. Secondly, we aim at producing diagnostics as short and intuitive as possible. We consider that the amount of statements in a diagnostic may be a factor compromising its understandability. Therefore, the representation adopted has to be as concise as possible, because the larger the representation is, the larger is the amount of difference diagnostic statements. Thus, an ideal representation would be one that is as close as possible to an $n \times n$ matrix but ensuring a well-accepted notion of equivalence.

2 Main definitions and results

As a starting point, we use event structures for representing the behavior of process models. Generally speaking, event structures are models of concurrency consisting of a set of events and behavioral relations between events. More specifically, our work relies on *prime* [3] and *flow event structures* [2], hereinafter abbreviated as PES and FES correspondingly, which are formally defined as follows:

Let E be a set of events and λ be a labeling function, such that λ associates each event to a label. The tuples $\mathbb{P} = \langle E, \leq, \#, \lambda \rangle$ and $\mathbb{F} = \langle E, \prec, \#, \lambda \rangle$ are **prime event structures** and **flow event structures**, respectively, where:

- $\leq$ is a partial order, known as *causal relation*, such that $\lfloor e \rfloor = \{e' \in E \mid e' \leq e\}$ is finite for all $e \in E$.
- $\#$ is a symmetric conflict relation. In the case of PES, $\#$ is irreflexive and hereditary with respect to causality, i.e., for all $e, e', e'' \in E$, if $e \# e' \prec e''$ then $e \# e''$.
- $\prec$ is an irreflexive non-transitive relation, known as the flow relation.

Figure 2 depicts the prime event structures of the processes in Figure 1. The arrows represent *causal* relations and the annotated dotted lines represent *conflict* relations. For the sake of simplicity, both transitive and hereditary relations were omitted in Figure 2. It can be noted that the PES of Figure 2(b), which corresponds to the process in Figure 1(b), contains two events with label c. The multiple occurrences of task c in the PES is due to the properties of the PES.

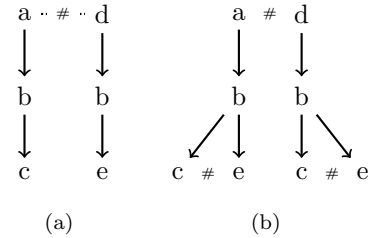


Figure 2: PES of processes in Fig. 1

FES are a more general type of event structures than PES. Indeed, every PES is also a FES [2]. The transformation of a PES into a FES is straightforward: the flow relation corresponds with transitive reduction of causality. Moreover, the conflict relation needs to be explicitly represented, as it is not longer hereditary because of the lack of transitive causality.

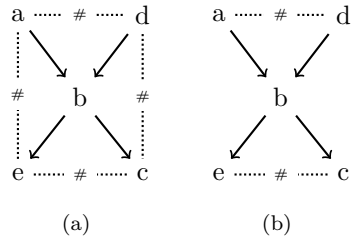


Figure 3: FES of processes in Fig.1

Due to the expressive power of flow event structures, multiple FES may exist to represent the same behavior, under *hp-bisimilarity* [4], a well-known notion of true concurrent equivalence for event structures. For instance, the FES depicted in Figure 3 represent the same behavior as those in Figure 2. In this context, our main contribution is the definition of a behavior-preserving folding operator, that allows us to reduce the size of the structure.

The aim of the operator is to find occurrences of the same task in a FES and replace them for only one occurrence, we refer to this set as *combinable set of events*. The defined folding operator establishes a set of restrictions that the combinable sets shall fulfill. Such restrictions are aligned to ensure the equivalence between a FES and its folded version according to hp-bisimilarity. The details about the folding operator, the restrictions for considering a set of events as combinable and proofs showing that the operator preserves hp-bisimulation are published in a technical report [1].

The folding operator can be applied repeatedly until no more combinable set of events is found. Ideally, the smaller is the event structure the more concise diagnostic would be. As a way of example, a diagnostic derived from the event structures in Figure 2 would contain at least 8 differences statements whereas the diagnostic derived from the event structures in Figure 3 would consist of only two statements. Indeed, from Figure 3 we can clearly see that *"tasks a and e never appear in the same run"*, whereas there is no restriction for a and e to appear in the same computation in (b), it can be interpreted as *"there is at least one run where tasks a and e can occur"*.

Although minimality seems an important property, canonicity is itself crucial for the purpose of model comparison. Unfortunately, the FES shown in Figure 4 provides a negative result in that respect: FES in Figure 4(b) and (c) are both hp-bisimilar to FES in Figure 4(a) and are minimal in size, but they are not isomorphic themselves. Therefore, further work is required to 1) determine if a canonical form can be properly defined and 2) determine the conditions to be observed during the folding to achieve the canonical form of an input FES.

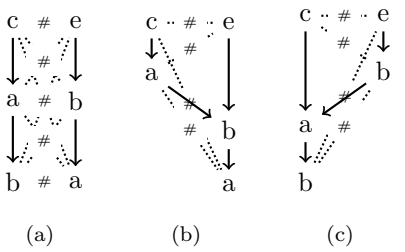


Figure 4: FES and two of its possible foldings

We foresee three major axes for further work: 1) Fully characterizing a canonical form for FES and refining our method to reduce the size of any input FES to its canonical form, 2) Extending our approach to cover the cases of process models with cycles, and 3) evaluating empirically the usefulness of diagnostics derived with our method with business analysts using real-world process models.

References

[1] Abel Armas-Cervantes, Luciano Garca-Bauelos, and Paolo Baldan. Behavioral comparison of acyclic business process models. Internal report, Institute of Computer Science, University of Tartu, 2013.

[2] Gérard Boudol and Ilaria Castellani. Permutation of transitions: An event structure semantics for ccs and sccs. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 411–427, London, UK, UK, 1989. Springer-Verlag.

[3] Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri Nets, Event Structures and Domains, Part I. *Theoretical Computer Science*, 13:85–108, 1981.

[4] Rob van Glabbeek and Ursula Goltz. Equivalence notions for concurrent systems and refinement of actions. In Antoni Kreczmar and Grazyna Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, volume 379 of *Lecture Notes in Computer Science*, pages 237–248. Springer Berlin Heidelberg, 1989.

[5] Matthias Weidlich, Jan Mendling, and Mathias Weske. Efficient Consistency Measurement Based on Behavioral Profiles of Process Models. *IEEE Transactions on Software Engineering and Methodology*, 37(3):410–429, 2011.

Translating a Modeling and Simulation Language to Hybrid Automata

Kevin Atkinson[1], Adam Duracz[2] and Walid Taha[1]

[1] Rice University and Halmstad University
{kevin.atkinson,walid.taha}@hh.se
[2] Halmstad University
adam.duracz@hh.se

1 Introduction

Tools such as SpaceEx [5] are pushing the limits of formal analysis for hybrid systems and today support models with up to a hundred variables. This still falls orders of magnitude short of what simulation tools are capable of, but it suggests that verification of complex hybrid systems models may be possible.

As the complexity of models increases, the expressiveness of the language in which they are written also becomes a concern. In this sense, verification tools still lag behind simulation tools, which often provide the user with a syntax closer to that of a programming language. Simulation tools are built around languages whose main objective is to allow the user to conveniently and accurately represent the key features of a system; for example, languages like Modelica [6] and Simscape [1] support abstraction through function definitions and object-oriented modeling. This amounts to a large feature set that tends to keep expanding, catering to the needs of domain experts wishing to express themselves efficiently.

Supporting such an expressive syntax poses a challenge to the verification community. Formal analysis requires well-understood and rigorously defined notions on which to operate, and every additional language construct increases the burden of the analysis. We can proceed in two ways to resolve this problem: either we express our analysis directly in terms of the wider syntax, effectively bringing the complexity of the surface language into our analysis; or we can opt for a translation of the wide syntax into a smaller one, in terms of which we can perform our analysis. This paper investigates the latter option by translating a simple modeling language (Acumen [10], shown in in Figure 1a) to hybrid automata [7] that can be formally analyzed [8]. Source code for the implementation of this transformation is available [2].

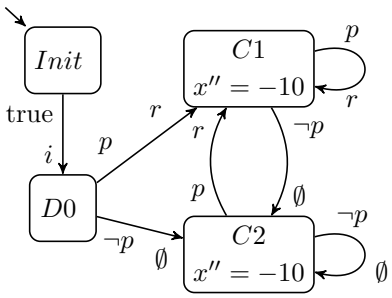
The Translation Challenge Acumen is a lightweight language for modeling hybrid systems. The continuous parts of the system are modeled by ordinary differential equations. The discrete parts are modeled by conditional statements and assignments, executed in a way that makes the order of statements irrelevant.

There are two kinds of interpreters available for the Acumen language: (1) those based on regular floating-point arithmetic that yield simulation results analogous to those obtainable using tools like Simulink [4] or OpenModelica [6]; and, (2) the Acumen enclosure interpreter [8] that yields guaranteed bounds on the simulation trajectory of the system, analogous to those produced by hybrid systems reachability analysis tools [5].

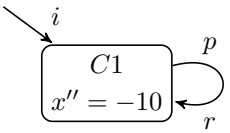
While the first kind of interpreter supports the full Acumen syntax, the enclosure interpreter supports a subset corresponding to a hybrid automaton, such as the one illustrated in Figure 1c. The goal of the algorithm described in Section 2 is to extend the syntax available to users wishing to analyze their model using the enclosure interpreter. For example, the extended syntax should permit writing models whose modes (in the sense of hybrid automata) are not given

```
class Main(simulator)
  private
    x  :=  5;
    x' :=  0;
    x'' := -10
  end
  if x <= 0 && x' <= 0
    x' := -x'/2
  end;
  x'' = -10
end
```

(a) Bouncing ball in Acumen



(b) After initial construction



(c) Final hybrid automaton

Figure 1: Input model, intermediate hybrid automaton and output hybrid automaton. In the above hybrid automata, p is the predicate $x \leq 0 \wedge x' \leq 0$, i is the reset $\{x := 5,\ x' := 0\}$, r is the reset $x' := x'/2$, and $\emptyset$ is the identity reset.

explicitly, and where hierarchies of hybrid behaviours can be expressed. This is accomplished by translating a wider syntax ($\mathcal{A}$), allowing arbitrary nesting of conditional statements intermixed with differential equations and assignments, to the hybrid automaton subset ($\mathcal{B}$).

2 Transformation Algorithm

To explain the basic steps of the algorithm, we use a simple Acumen program, modeling the bouncing ball hybrid system [8], which is shown in Figure 1a. The transformation of this model into a hybrid automaton is simple, but still involves identifying modes and transitions, as this information is not given explicitly in the model. More advanced models require additional passes that due to space are not included in this abstract.

The Basic Algorithm The algorithm starts out by reducing the program into a normal form, consisting of a sequence of `if` statements. Empty `else` branches are first added to any `if` statement lacking an `else` branch and then all `else` branches are converted into `if` statements (with negated predicates). Assignment statements and equations are pushed down into the most nested conditional statements and the conditionals are converted into top-level `if` statements, with predicates corresponding to their path condition.

Each top-level `if` statement C subsequently generates a mode M_C and a transition T_C. The continuous assignment statements of C become the differential equations of the mode M_C. The condition of C becomes the guard of T_C, and the discrete assignments of C, together with the additional discrete assignment $mode := M_C$, become the reset of T_C. The additional discrete assignment encodes the target mode of the transitions in the automaton and is a key step in this algorithm. The source mode of each T_C is derived from its guard.

Two initial modes, *Init* and $D0$, are also added. *Init* serves as the source mode of a transition whose reset corresponds to the initial values of the system. Because the initial mode of the output automaton has not yet been resolved at this stage, a default initial mode $D0$ is added.

The preceeding steps result in the automaton shown in Figure 1b. Additional passes are then used to obtain a more minimal automaton. The first pass merges modes with identical sets of differential equations and out-going transitions. In this pass, $C2$ will merge into $C1$. The second pass attempts to eliminate unnecessary modes containing a transition that must trigger and has a target mode other than its source. In this pass, $D0$ is eliminated, leaving only *Init* and $C1$. The resets in the transition from the *Init* mode become the initial conditions of the automaton leaving only mode $C1$.

Advanced Passes Handling additional language constructs in the source model requires extending the translation with more advanced passes. For example, when the model contains a switch statement, the translation may need to eliminate the variable used to switch between case clauses. To do this, the translation relies on an analysis based on postcondition and precondition predicates that are associated with all transitions and modes, respectively. The source mode of a transition is identified by comparing the precondition of each mode with the guard of the transition. The target mode is defined by matching the postcondition of the transition with the precondition of the mode. Concretely, the target mode of the transition T_C, with reset $mode := M_C$ (inducing a component $mode = M_C$ into the postcondition of T_C), is the mode M_C whose precondition includes $mode = M_C$ as a component.

3 Related Work

Translations from programming language like formalisms to hybrid automata, with the aim of applying formal methods to the source programs, have been studied previously. The approach described by Agrawal, Simon and Karsai [3] is based on a translation of models expressed in Simulink/Stateflow [4] through intermediate languages. Lyde and Might [9] describe the translation of an extended core calculus for the MATLAB programming language, to a subset of Scheme. We are concerned with the translation of a subset of the Acumen language, down to a smaller subset directly corresponding to a hybrid automaton, to enable analysis using the Acumen enclosure interpreter [8]. Crucially, the source programs (in syntax $\mathcal{A}$) that we aim to support do not specify modes and events explicitly.

References

[1] Mathworks. Simscape. `http://www.mathworks.com/products/simscape/`.

[2] Translation algorithm as part of the Acumen source code repository (*extract* branch). `https://bitbucket.org/effective/acumen-dev/branch/extract`.

[3] A. Agrawal, G. Simon, and G. Karsai. Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations. In *International Workshop on Graph Transformation and Visual Modeling Techniques*, 2004.

[4] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Design Autom.*, 2006.

[5] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *International Conference on Computer Aided Verification*, LNCS, 2011.

[6] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nystrom, L. Saldamli, D. Broman, and A. Sandholm. OpenModelica - a free open-source environment for system modeling, simulation, and teaching. In *IEEE International Symposium on Intelligent Control*, 2006.

[7] T. A. Henzinger. The theory of hybrid automata. IEEE Computer Society Press, 1996.

[8] M. Konecny, W. Taha, J. Duracz, A. Duracz, and A. Ames. Enclosing the behavior of a hybrid system up to and beyond a zeno point. In *IEEE International Conference on Cyber-Physical Systems, Networks, and Applications*, 2013.

[9] S. Lyde and M. Might. Extracting hybrid automata from control code. In *NASA Formal Methods*, 2013.

[10] W. Taha, P. Brauner, Y. Zeng, R. Cartwright, V. Gaspes, A. Ames, and A. Chapoutot. A core language for executable models of cyber-physical systems (preliminary report). In *International Conference on Distributed Computing Systems Workshops*. IEEE Computer Society, 2012.

Input-Output Conformance Testing of Software Product Lines

Harsh Beohar and Mohammad Reza Mousavi

Center for Research on Embedded Systems, Halmstad University, Sweden
harsh.beohar@hh.se, m.r.mousavi@hh.se

1 Introduction

Software Product Lines (SPLs) have become common practice in software development and have been proven effective in mass production and customization of software. There have been several attempts to provide a structured discipline for testing SPLs. Furthermore, composing test suites in a structured way is also studied in different areas of software engineering. For instance, in [10], the authors present a model-based approach to test aspect-oriented programs. However, it appears from recent surveys [3, 4, 7, 6] that several fundamental approaches to model-based testing (based on finite state machines and labeled transition systems) are not yet fully adapted to and adopted in this domain.

In this abstract, we propose to adopt Input-Output Featured Transition Systems (IOFTSs) as fundamental and expressive models for model-based testing of SPLs. To this end, we adapt the traditional Input-Output Conformance (IOCO) theory [11] to allow for using IOFTSs as test models.

We define a notion of the test suite and the set of test cases generated from an IOFTS, which can be used for checking conformance. We define two notions of refinement, one at the level of IOFTSs and another one at the level of test suites, that allow for focusing on particular sets of features and eventually on a particular product. We show that these two refinements interact nicely, in that they lead to the same set of test cases.

This abstract is organized as follows. In Section 2, the notions of IOFTS and product derivation are explained informally. Section 3 provides a brief overview of our main results. In Section 4, some open issues for future research are outlined. Due to the space restriction, we only present a brief overview of our approach. We refer to [1] for a precise and detailed treatment of the approach.

2 Background

Feature diagrams [5, 9] have been used to model variability constraints in SPLs using a graphical notation. A feature diagram represents all valid products of an SPL in terms of features that are arranged hierarchically. Usually, feature diagrams are represented by a directed acyclic graph, of which each node is a feature. There are different kinds of edges between a parent node (feature) and its children (sub-features), namely, the ones representing the *mandatory* sub-features, and the others representing the *optional* sub-features. In addition, a feature diagram can specify extra constraints on features; namely, the *alternative* relationship, the *exclude* relationship, and the *require* relationship.

A feature diagram only specifies the structural aspects of variability in an SPL; however, to formally analyze the behavior of an SPL, we follow the approach of [2] in annotating the transitions of a labeled transition system with logical constraints on the presence or absence

of features; the features used in such logical constraints are assumed to be already specified in a feature diagram. We slightly differ from [2] by distinguishing the alphabets of the labeled transition systems into two disjoint sets of inputs and outputs. This is a necessary ingredient for extending the theories of testing, and particularly IOCO, to this setting. In short, a *input-output feature transition system* (IOFTS) consists of a *input-output labeled transition system*, a *feature diagram*, a *feature annotation* function, and a set of *product configurations* representing the set of valid products induced by a feature diagram.

Subsequently, we define a family of product derivation operators (parameterized by feature constraints), which project the behavior of an IOFTS into another IOFTS representing a selection of products (a product sub-line). Using such representations as test models different products of an SPL can be analyzed simultaneously.

3 Results

Given a specification modeled as an IOFTS and assuming that an implementation under test can be expressed as an (unkown) IOFTS (similar to the testing assumption of [11]), it is possible to define an Input-Output Conformance (IOCO) relation between the two. Intuitively, the defined IOCO relation asserts that the experiments derived from a specification and executed on the implementation under test, results in outputs that are always allowed by the specification. This corresponds exactly to the extensional definition of IOCO on labelled transition systems [11].

To complement the intensional definition, we give an operational definition of test suites and test cases, which can also be expressed as IOFTSs derived from a given specification. Moreover, we define a notion of refinement that projects a test-suite into the part that satisfies a certain feature constraint. This allows us to generate a test suite for a product line and refine it into test suites for more specific sub-lines (and eventually generating test cases for a specific product). Furthermore, we show that the two notions of refinement (one at the specification level and the other at the test suite level) are consistent. In particular, we showed that by refining a test suite of a specification, we obtain a test suite that is isomorphic to the test suite generated from the refined specification (assuming that both refinements use the same feature constraint).

4 Open issues

In future, we would like to research the following open issues.

1. *Factoring test suites.* The main goal of this research line is to provide a theory of SPL testing that allows for testing common features among different products once and for all. As a first step to this end, we intend to define an operator that given two models (or two test suites) with different feature constraints, returns a test suite, which represents the common features of the two models and two test suits that cover the specific features of each of the two models.

2. *Incremental testing.* Our refinement operators (both at the level of specification and test suite) are top-down in nature, i.e., these operators refine the behavior of an abstract specification (test suite) by strengthening the associated feature constraint. Conversely, it is also possible to perform testing in a bottom-up manner, where the behavior of concrete products are combined to validate an SPL and the test suite of an SPL is generated compositionally.

3. *Empirical research.* Lastly, we would like to implement our theoretical framework and perform empirical research on its effectiveness and efficiency.

References

[1] H. Beohar and M.R. Mousavi. Input-output conformance testing based on featured transition systems. Submitted for publication, August 2013. Available from: `http://ceres.hh.se/mediawiki/images/0/0c/Fioco_2013.pdf`.

[2] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking (to appear). *IEEE Trans Software Eng (TSE)*, 2012.

[3] P.A. da Mota Silveira Neto, I. do Carmo Machado, J.D. McGregor, E.S. de Almeida, and S.R. de Lemos Meira. A systematic mapping study of software product lines testing. *Inf. Softw. Technol.*, 53(5):407–423, 2011.

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

[5] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[6] B.P. Lamancha, M.P. Usaola, and M.P. Velthius. Systematic review on software product line testing. In *Software and Data Technologies*, volume 170 of *Comm. in Computer and Information Science*, pages 58–71. Springer, 2013.

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

[8] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[9] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *Proc. of the 14th IEEE International Conference on Requirements Engineering*, pages 136–145. IEEE, 2006.

[10] W. Xu and D. Xu. A Model-Based Approach to Test Generation for Aspect-Oriented Programs. In *Proc. of the 1st Workshop on Testing Aspect-Oriented Programs*, ACM. Chicago, 2005.

[11] J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 1–38. Springer, 2008.

Formalizing a System for Deadlock Checking by Data Race Detection in Coq

Peter Brottveit Bock

Department of Informatics, University of Oslo

Creating concurrent programs is considered more difficult than single threaded programs. The two most common classes of errors are *deadlocks* and *race conditions*, where deadlocks occur when several threads are waiting in a cyclic manner for resources that other holds, whereas race conditions happen when several threads access a shared variable without protection with where at least one thread modifying the variable.

It is therefore desirable to statically analyze programs to check if they contain deadlocks. The systems which do this are not trivial; they usually contain many definitions and tedious proofs. If such a system is presented, is it therefore of interest to formally check that the system has the properties it has been claimed to have.

This abstract describes the formalization of a set of type and effect system related to deadlock checking, using the tools Ott[1] and Coq.

1 Background

In the technical report *Deadlock Checking by Data Race Detection* [2], a novel idea is introduced: a reduction from deadlock checking to data race checking. The idea is that given a program with locks, it can be analyzed to approximate how the locks are used, and then fresh variables can be inserted into the code at strategic locations. If there is a deadlock in the original program, the inserted variables will cause a race condition. One can therefore leverage current and future tools and research about data race detection, into deadlock checking. For race condition detection, there exists powerful static analysis tools, such at Goblint for C and Chord for Java.

The language of study is a functional language, with dynamic thread creation, dynamic lock creation, and reentrant lock acquiring.

The semantics is specified by a small-step operational semantics, divided into two parts: *local* steps and *global* steps. The local steps are syntax-directed and concern how a single thread operates independent of other concurrent threads. The global steps describe spawning of threads, manipulation of locks, together with interleaving of the local steps for a finite set of threads.

The analysis centers around estimating an upper bound for how many times a lock is acquired by a thread, and how this is changed by executing an expression. The latter is called an *effect* of the expression. An estimation is denoted by Δ, which can be considered a mapping from lock set variables to $\mathbb{R} \cup \{\pm\infty\}$. Finally, $t :: \Delta \to \Delta'$ means that executing t from the initial, abstract state Δ, will lead to the state Δ' (as a partial correctness property).

Also function types are annotated with effectw, as in $\hat{T} \xrightarrow{\Delta \to \Delta'} \hat{T}'$, which means that the body of the function has the effect $\Delta \to \Delta'$. Additionally, lock types are annotated with lock sets, indicating where in the program a lock may have been created.

The first type system is the specification, and the purpose of it is to state the desired system in a clear, elegant and declarative manner. It has judgments on the form

$$C; \Gamma \vdash t : \hat{S} :: \Delta_1 \to \Delta_2,$$

where C is a set of constraints, Γ is the context, mapping variables to type schemes, t is a single thread, and $\hat{S}$ is an annotated type scheme.

As the purpose of this system is to get a clear understanding of what is wanted, the system is not syntax-directed, and non-deterministic. The downside is that it does not make it clear whether the analysis can be done efficiently. This leads us to the next type system.

The algorithm is a syntax-directed type system without non-determinism. The most important difference from the specification, is that the algorithm *generates* a set of constraints. The form of a judgment is therefore $\Gamma \vdash t : \hat{T} :: \Delta_1 \to \Delta_2; C$.

Another thing to note is that the judgment has a type in the conclusion, not a type scheme. The reason is that, while the specification freely allows instantiation and generalization of a type at any point, the algorithm does this only at the places where it is necessary (instantiating when referencing a variable, generalizing in let-expressions).

Several properties are proved informally: subject reduction between the syntax-directed system and the semantics; soundness of the algorithm with regards to the specification; and completeness of the algorithm with regards to the syntax-directed system.

2 The Formalization

My work consists of a formalization of the syntax, semantics, type systems, and the proposition and proof of soundness. The goal of the formalization is to make the definitions and properties machine checkable. The proofs have been done top-down, and are cut off when they reach "obvious" truths.

The tools used to achieve this are Ott, a tool specialized for programming language theory researchers, and Coq, a very powerful, general purpose theorem prover assistant.

The grammar and most judgments have been formalized in Ott. Ott naturally lends itself to a deep embedding, but it is possible to bypass Ott's generation of Coq-code for the abstract syntax tree, and define manually how a grammar should be translated to Coq. Bypassing Ott's manual generation of Coq-code make the embedding shallower, and one can leverage existing theorems.

Ott can generate substitution and free variable functions from the grammar. Unfortunately, using some of the more advanced features of Ott makes the generated functions useless and they have to be hand coded in Coq. Additionally, the substitution functions generated are not capture avoiding.

The judgments from the paper can fairly directly be translated to Ott. Properties which are not given as syntactical rules, are coded by hand in Coq.

One initial challenge was the syntax of a program, which casually states that the $||$-operator should be associative and commutative, with $\emptyset$ as the identity. This property should, presumably not be used in the algorithm, as this would cause non-determinism in how to apply the rules. The current proofs have not needed this property yet, but it could be needed in either a sub-proof or in a later proof.

One of the major issues was how to handle *freshness*-claims in derivations, which are usually given as a premise *X is fresh*. Such a claim is not local; to check if it holds, the whole derivation must be inspected. The workaround needed to solve this are not pretty. Two ways it can be solved are:

1. Let the claim trivially be solved by a constructor, and then define a proposition on derivation which checks that the freshness-claims actually hold.

2. Modify the rules, so that information about all variables used "flow" through the system, so that freshness can be checked locally.

The first solution has the nice property that the rules will be similar to those presented in the paper, but it also forces the derivation to live in Coq's *Type*, and provability then becomes both inhabitance and the truth of a predicate on derivations.

To implement the second solution, the rules have to be littered with variables which makes it possible to generate fresh variables, and to enforce that these generated variables are not used. Since variables are indexed by numbers, it is natural to make each rule take a number as its "input", which can be incremented to generate fresh variables, and "return" the new maximum number. This is the chosen solution.

Formalization of the soundness proof has been reduced to simpler lemmas. During the process, errors of varying degrees of importance was found. Some related to weaknesses in the formalization, others to properties not mentioned in the paper, and finally technical errors. No error found has been grave enough to dismiss the general idea, but especially the typing rules for recursive functions are problematic.

3 Experience

Using Ott alone is useful, as it can detect ambiguity, and since it generates both LaTeX and Coq code, one never ends up with inconsistencies between the definitions in a paper and in the theorem prover. Also, the possibility to print out the grammars and judgments in an accessible notation makes it possible to discuss the work with people not familiar with the syntax of Coq.

The downsides with Ott is that the code generation can feel unreliable, and that the generated function for free variables and substitutions often become useless.

As someone with some experience with Coq, Ott forces a perspective where syntax and judgments are first class citizens, while functions and general propositions are second class. When a Ott-project grows larger, the idiosyncrasies of Ott come forth, and problems for which one has a straightforward Coq-solution need to be worked into language of Ott.

References

[1] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. *Ott: Effective Tool Support for the Working Semanticist.* ICFP, 2007.

[2] Ka I Pun, Martin Steffen, Volker Stolz, *Deadlock Checking by Data Race Detection.* Preprint submitted to Elsevier, 2013.

Inheritance *Is* Subtyping

Robert Cartwright[1,2] and Moez Abdel-Gawad[1]

[1] Rice University, Houston, TX, USA
cork@rice.edu, moez@rice.edu
[2] Halmstad University, Halmstad, Sweden
robert.cartwright@hh.se

Since Luca Cardelli wrote a seminal paper [3] on the semantics of inheritance in 1984, programming language researchers have constructed a variety of structural models of object-oriented programming (OOP) founded on Cardelli's work. Since Cardelli approached OOP from the perspective of functional programming, he identified inheritance with record subtyping—an elegant choice in this context. Although Cardelli did not formally define inheritance, he equated it with record extension and proved that for a small functional language with records, variants, and function types–but no recursive record types–that syntactic and semantic record subtyping were equivalent. William Cook *et al* [4] subsequently added `ThisType` and recursive record types, narrowing the typing of **this** in inherited methods, and reached a profoundly different conclusion: *inheritance is not subtyping.*

Meanwhile, object-oriented (OO) program design emerged as an active area of research within software engineering, spawning class-based OO languages like C++, Java, and C#, which strictly define inheritance in terms of class hierarchies. In these languages, subtyping is identified with inheritance. In contrast to Cardelli's expansive formulation of inheritance based solely on record interfaces (sets of member-name interface pairs)[1], these languages define the type associated with a class C as the set of all instances of C and all instances of explicitly declared subclasses of C. Simply matching the signatures of the members of C–as in record subtyping–is insufficient.

It is easy to show that record subtyping is too weak to capture the notion of inheritance in nominally typed OO languages. Consider classes `Set` and `MultiSet` intended to represent mathematical sets and multi-sets respectively. They can easily have exactly the same visible members with exactly the same types. Assume that `Set` and `MultiSet` are defined independently but have exactly the same visible members. If objects simply denote records, then each class is a subtype of the other, but in a nominal OO language, no such subtyping relationship exists between the two classes. In nominal OO languages, objects are more than mere records.

In Cardelli's semantics and its successors based on functional programming models, the meaning of a class only depends on the members of the class (including *inherited* members), not on the inheritance hierarchy used to define the class. This paper dicusses the implications of a new approach to defining the semantics of OO languages that embeds in each object the signature of the inheritance hierarchy above it. In contrast to record-based semantics, our new approach completely reconciles inheritance and subtyping among classes: a class B is a subtype of a class A iff B inherits from A.

In statically-typed functional languages based on the simply typed lambda-calculus, the issue of subtyping does not arise: every data value belongs to a unique type. Even when such a language is generalized to support parametric polymorphism [9], every value belongs to a unique *monotype* (unquantified type).

Object-oriented languages introduce the idea that composite values (often called records or structures in functional languages) can belong to multiple monotypes. For example, a

[1] Since Cardelli excluded recursive types, every interface in his language can be expressed purely in terms of type constructors applied to primitive types.

`ColorPoint` object with fields `x: Number, y: Number, color: Color` can have type `Point` which omits the `color` field as well as type `ColorPoint`. In *structural* OO languages, object types are based simply on the interfaces of objects: the names and types of their visible record members. Hence, `ColorPoint` is a subtype of `Point` even when it is separately defined without use of inheritance. In *nominal* OO languages, object types are based strictly on the inheritance structure specified in the program: the type associated with class `B` is a subtype of the type associated with class `A` iff the definition of `B` explicitly inherits from the definition of `A`. If object values do not include inheritance information, this restricted definition of subtyping appears capricious. But OO software developers think of an object in the context of its class hierarchy and the contracts associated with its class members, *which are inherited along with the corresponding class members*. For example, in Java, the interface `Comparable<T>`, consisting only of the method `int compareTo(T t)`, has a contract asserting that `compareTo` defines a total ordering on `T`; an arbitrary class with a method `int compareTo(T t)` generally does not obey this contract. When a programmer asserts that a class `C` implements `Comparable<C>`, he is asserting that the `compareTo` defines a total ordering on `T`.

In mainstream OO design, subtyping conforms to the *Contract Presevation* (CP) property: types are characterized by behavioral contracts and every subtype `B` of a type `A` obeys the contracts of the parent type `A`. The earliest formulation of this principle, proposed by Liskov in 1988 [7], was expressed in terms of the substitutability of objects, which was technically problematic, perhaps dissuading researchers in programming theory from giving it much credence. The principle is still called the *Liskov Substitution* principle in most software engineering contexts. A subtype can augment the contracts inherited from its parent type, but the inherited contracts still apply as well. Of course, no decidable type system can fully capture program behavior since any non-trivial aspect of program *behavior* is undecidable by Rice's theorem. In practice, a decidable type system should perform static checks that help programmers confirm that their code obeys CP.

In nominal OO languages like Java and C#, the static type system identifies subtyping with inheritance: the type corresponding to class `C` consists of all instances of `C` and all instances of subclasses of `C`, which by definition inherit from `C`. Hence, the type for class `B` is a subtype of the type for class `A` iff `B` inherits from `A`. In writing the code for a subclass, the programmer is responsible for confirming that instances of the class conform to the contracts for all superclasses. The preservation of such contracts is a pillar of good OO design. For this reason and to support mutability of object fields, the input types in the signature of an overriding method typically must exactly match those in the overridden method.

According to folklore among programming language researchers, the identification of subtyping and inheritance in mainstream OO languages like Java and C# is misguided, despite the fact that simple versions of these type systems have been proved sound [5, 6] relative to operational semantics for these languages. In earlier work [1], we presented denotational model of OOP, dubbed **NOOP**, akin to Cardelli's record model that justifies the typing conventions in mainstream OO languages and breaks typing rules based on record subtyping. In other words, given what we believe is a proper model of mainstream nominal OOP, *subtyping is inheritance* and the usual structural typing rules are *unsound*.

The key idea in the construction of **NOOP** is to define the meaning of an object in class `C` as a pair consisting of a record (as in structural models) plus the signature closure for class `C`. A *signature* s is a triple consisting of a *key* class name `C`, a (possibly empty) set of superclass names, and a set of syntactic types for the visible members of `C`. A *signature environment* se is a finite set of class signatures where no two key class names are the same. Hence, a signature environment determines a function mapping a finite set of class names to signatures. A signature

environment is *closed* iff every class name that appears anywhere in the environment appears as a key class name. In other words, every class that is referenced in the environment is defined in the finite function corresponding to the environment. A signature environment *se* is a *signature closure for class name* `C` iff (*i*) *se* is closed, (*ii*) *se* defines the name `C`, and (*iii*) *se* only defines the names in the transitive reference closure of `C`.

From the preceding definitions, we deduce that a signature closure for class name `C` is a signature environment consisting of the transitive closure (under class reference) of the singleton set consisting of a signature for class name `C`.

For the details on how to construct **NOOP**, see [1] and [2].

It is straightforward to define a syntactic relation between class signatures called subsigning: the class signature s_1 subsigns the class signature for s_2 iff the information in s_1 includes the information in s_2 and both signatures are well-formed. The details of the construction are given in [1]. We identify subsigning with inheritance. In a nominal OO program `P`, class `B` inherits from class `A` iff the signature closure for `A` subsigns the signature closure for `B`.

For each class `C` in a program `P`, we can define the weak ideal [8] of objects consisting of all objects with signature matching the signature of C in P and all objects in classes with signatures that subsign the signature of `C`. This weak ideal includes the instances of *all possible classes* extending `C`. Moreover, it is not difficult to prove that subsigning implies subtyping. The proof of this property appears in [1]. Hence, in nominal OO languages, inheritance implies subtyping.

References

[1] Moez A. AbdelGawad. A domain-theoretic model of nominally-typed object-oriented programming. *Accepted for publication in the Journal of Electronic Notes in Theoretical Computer Science (ENTCS). Also presented at The 6th International Symposium of Domain Theory and Its Applications (ISDT'13)*, 2013.

[2] Moez A. AbdelGawad. *NOOP: A Nominal Mathematical Model Of Object-Oriented Programming.* Scholar's Press, 2013.

[3] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[4] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 125–135, New York, NY, USA, 1990. ACM.

[5] Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the java type system sound? *Theor. Pract. Object Syst.*, 5(1):3–24, January 1999.

[6] A. Igarashi, B. Pierce, and P. Wadler. Featherweight java: A minimal core calculus for java and gj. In *OOPSLA*, 1999.

[7] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994.

[8] David Macqueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphism. *Information and Control*, pages 95–130, 1986.

[9] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

The Delay Monad and Restriction Categories

James Chapman, Tarmo Uustalu and Niccolò Veltri

Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia
{james,tarmo,niccolo}@cs.ioc.ee

Restriction categories are an abstract axiomatic framework by Cockett and Lack for reasoning about partiality of functions [3]. In a restriction category, every map $f : A \to B$ is required to define an endomap $\overline{f} : A \to A$, satisfying four equational conditions:

$$\begin{aligned} &R1. \quad f \circ \overline{f} = f \\ &R2. \quad \overline{g} \circ \overline{f} = \overline{f} \circ \overline{g} \\ &R3. \quad \overline{g} \circ \overline{f} = \overline{g \circ \overline{f}} \\ &R4. \quad \overline{g} \circ f = f \circ \overline{g \circ f} \end{aligned}$$

A map f is called total, if $\overline{f} = \mathrm{id}$. The map $\overline{f}$ should be thought of as a partial identity function on A specifying the domain of definedness of f. The restriction operator also defines a partial order on maps, $f \leq g$ if and only if $f = g \circ \overline{f}$. That is, f is less defined than g if f coincides with g on f's domain of definedness.

Restriction categories are related to partial map categories, which are the classical synthetic approach to partiality: every partial map category is a restriction category and a restriction category is a partial map category intuitively whenever the restriction category contains as objects all the domains of definedness. Formally this is when all restriction idempotents split.

An example of a restriction category particularly relevant for dependently type programming is the Kleisli category of the delay monad. The delay monad was introduced by Capretta [2] as a means to incorporate general recursion to type theory and it is useful in this setting for modeling non-terminating behaviours. It constitutes a constructive alternative to the maybe monad. For a given type A, each element of $\mathsf{Delay}\ A$ is a possibly infinite computation that returns a value of A, if it terminates. We define $\mathsf{Delay}\ A$ as a coinductive type by the rules

$$\frac{}{\mathsf{now}\ a : \mathsf{Delay}\ A} \qquad \frac{c : \mathsf{Delay}\ A}{\mathsf{later}\ c : \mathsf{Delay}\ A}$$

Propositional equality is not suitable for coinductive types. We need different notions of equality, namely strong and weak bisimilarity. Two computations are strongly bisimilar, if they contain the same number of applications of later, i.e., it takes the same (possibly infinite) amount of time for them to converge to the same value. Strong bisimilarity is defined coinductively by the rules

$$\frac{}{\mathsf{now}\ a \sim \mathsf{now}\ a} \qquad \frac{c \sim c'}{\mathsf{later}\ c \sim \mathsf{later}\ c'}$$

Weak bisimilarity is defined in terms of convergence. This binary relation between $\mathsf{Delay}\ A$ and A relates a terminating computation to its value and is inductively defined by the rules

$$\frac{}{\mathsf{now}\ a \downarrow a} \qquad \frac{c \downarrow a}{\mathsf{later}\ c \downarrow a}$$

Two computations are weakly bisimilar if they differ by a finite number of application of the constructor later, i.e., they either converge to the same value or diverge. Weak bisimilarity is defined coinductively by the rules

$$\frac{c \downarrow a \quad c' \downarrow a}{c \approx c'} \qquad \frac{c \approx c'}{\mathsf{later}\ c \approx \mathsf{later}\ c'}$$

The functor Delay quotiented by strong/weak bisimilarity is a strong monad. The Kleisli category of the delay monad quotiented by weak bisimilarity ($\mathbf{Kl}(\mathsf{Delay}/\approx)$) is a restriction category. The restriction $\overline{f} : A \to \mathsf{Delay}\ A$ of a map $f : A \to \mathsf{Delay}\ B$ is given in terms of the strength σ of Delay by

$$\overline{f} = \ A \xrightarrow{\langle \mathrm{id}, f \rangle} A \times \mathsf{Delay}\ B \xrightarrow{\sigma_{A,B}} \mathsf{Delay}\ (A \times B) \xrightarrow{\mathsf{Delay}\ \pi_0} \mathsf{Delay}\ A$$

The restriction category $\mathbf{Kl}(\mathsf{Delay}/\approx)$ has a rich structure that makes it suitable for analyzing computability. It is a restriction category with a partial final object and partial binary products, with joins of compatible maps, meets of maps with semidecidable codomains and a uniform iteration operator.

The partial final object is 1 and the unique good map from an object A into 1 is $\mathsf{now} \circ !_A$. The partial product of A and B is $A \times B$ with projections $\mathsf{now} \circ \pi_0$ and $\mathsf{now} \circ \pi_1$. The pairing operation is defined on computations first and then extended pointwise to maps. It runs the two computations (e.g., sequentially) and returns a pair of values a, b when both computations have terminated with values a and b respectively.

The joins and meets in $\mathbf{Kl}(\mathsf{Delay}/\approx)$ are given by pointwise extensions to maps of joins and meets of compatible computations and computations over semidecidable types. Intuitively one runs the two given computations in parallel. The join of two computations returns the value of the quicker one of them (if at least one of the two computations terminates at all), while the meet terminates only when both have terminated, provided they gave the same value, else it goes on forever.

The category $\mathbf{Kl}(\mathsf{Delay}/\approx)$ also supports a uniform iteration operator $\mathsf{iter} : (A \to \mathsf{Delay}\ (A + B)) \to A \to \mathsf{Delay}\ B$. Informally, iteration is defined as follows. We apply a given map f to an initial value a. When $f\ a$ converges to an element of B, iteration has converged to that element. When it converges to an element of A, it is time to apply f again to that element.

We have formalized all of the development above in the dependently typed programming language Agda [1].

We are interested in learning more about the structure of $\mathbf{Kl}(\mathsf{Delay}/\approx)$. In particular we would like to show that it is a Turing category.

Acknowledgement This research was supported by the ERDF funded Estonian ICT national programme project "Coinduction", the Estonian Science Foundation grants No. 9219 and 9475 and the Estonian Ministry of Education and Research target-financed research theme No. 0140007s12.

References

[1] A. Bove, P. Dybjer, U. Norell. A brief overview of Agda, a functional language with dependent types. In *Proc. of TPHOLs 2009*, v. 5674 of *LNCS*, pp. 73–78. Springer, 2009.

[2] V. Capretta. General recursion via coinductive types. *Log. Meth. in Comput. Sci.*, v. 1, n. 2, article 1, 2005.

[3] J. R. B. Cockett and S. Lack. Restriction categories I: categories of partial maps. *Theor. Comput. Sci.*, v. 270, n. 1–2, pp. 223–259, 2002.

A Comparison of Runtime Assertion Checking and Theorem Proving for Concurrent and Distributed Systems

Crystal Chang Din[1], Richard Bubel[2] and Olaf Owe[1]

[1] University of Oslo, Norway
[2] Technische Universitat Darmstädt, Germany
{crystald,olaf}@ifi.uio.no, bubel@cs.tu-darmstadt.de

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

Object orientation is the leading framework for concurrent and distributed systems, recommended by the RM-ODP [15]. However, method-based communication between concurrent units may cause busy-waiting, as in the case of remote and synchronous method invocation, e.g., Java RMI [2]. Concurrent objects communicating by *asynchronous method calls* have been proposed as a promising framework to combine object-orientation and distribution in a natural manner. Each concurrent object encapsulates its own state and processor, and internal interference is avoided as at most one process is executing on an object at a time. Asynchronous method calls allow the caller to continue with its own activity without blocking while waiting for the reply, and a method call leads to a new process on the called object. The notion of *futures* [6, 19, 12, 20] improves this setting by providing a decoupling of the process invoking a method and the process reading the returned value. By sharing *future identities*, the caller enables other objects to wait for method results. However, futures complicate program analysis since programs become more involved compared to semantics with traditional method calls, and in particular local reasoning is a challenge. ABS [17] is a high-level imperative object-oriented modeling language, based on the concurrency and synchronization model of Creol [18]. It supports futures and concurrent objects with an asynchronous communication model suitable for loosely coupled objects in a distributed setting. In this work, we present our testing and verification tools for ABS programs.

The execution of a distributed system can be represented by its *communication history* or *trace*; i.e., the sequence of observable communication events between system components [8, 14]. At any point in time the communication history abstractly captures the system state [10, 9]. In fact, traces are used in the semantics for full abstraction results (e.g., [16, 1]). The *local history* of an object reflects the communication visible to that object, i.e., between the object and its surroundings. A system may be specified by the finite initial segments of its communication histories, and a *history invariant* is a predicate which holds for all finite sequences in the set of possible histories, expressing safety properties [5].

In our reasoning system, we formalize object communication by an operational semantics

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

In this work we implement a runtime assertion checker and extend the KeY theorem prover for testing and verifying ABS programs, respectively. For runtime assertion checking the ABS interpreter is augmented by an explicit representation of the global history, reflecting all events that have occurred in the execution. And the ABS modeling language is extended with method annotations such that users can define software behavioral specification [13], i.e., invariants, preconditions, assertions and postconditions, inline with the code. We provide the ability to specify both state-based and history-based properties, which are checked during simulation. History wellformedness, i.e. the order of the events, the non-nullness of the calling objects and the characteristic of futures, is proved and need not be checked during execution. Although by using runtime assertion checking, we gain confidence in the quality of programs, correctness of the software is still not fully guaranteed for all runs. Formal verification may instead show that a program is correct by proving that the code satisfies a given specification. As formal verification tool we use and extend a variant of the KeY verification system [7], which supports ABS as target language. In particular, KeY features a semi-automatic theorem prover based on dynamic logic. The design of its Gentzen-style sequent calculus follows the symbolic execution paradigm. For the ABS formalisation in dynamic logic, we follow the approach developed in [4, 3], but use the improved history formalisation as presented in [11]. The characteristic feature of the calculus is that it achieves to stay in a sequential setting while reasoning about properties of concurrent and distributed systems.

ABS runtime assertion checking and theorem proving of ABS programs in KeY are illustrated via two examples: a fair version of the reader/writer example and a publisher/subscriber example. The first example shows how we verify the class implementation by relating the objects state with the communication history. The second example shows how we achieve compositional reasoning by proving the order of the local history events for each object. Along these two examples, we evaluate and compare both approaches with respect to their scope and ease of application. In particular, we investigate their strengths and weaknesses concerning the different properties. We give recommendations on which approach is suitable for which purpose as well as the implied costs and benefits of each approach. Finally, we identify areas where improvements are needed and provide directions of future research.

References

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

[2] A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. Theoretical Computer Science, 389(3):341 – 410, 2007.

[3] W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In K. Breitman and A. Cavalcanti, editors, *Proc. International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of Lecture Notes in Computer Science, pages 387–406. Springer, 2009.

[4] W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. Science of Computer Programming, 77(12):1289–1309, Oct. 2012.

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

[6] H. G. Baker Jr. and C. Hewitt. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 symposium on Artificial intelligence and programming languages*, pages 55–59, New York, NY, USA, 1977. ACM.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of Lecture Notes in Computer Science. Springer, 2007.

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

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

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

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

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

[13] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, June 2012.

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

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

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

[17] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of Lecture Notes in Computer Science, pages 142–164. Springer, 2011.

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

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

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

Towards Practical Verification of Dynamically Typed Programs

Björn Engelmann

Department of Computing Science, University of Oldenburg, Germany
bjoern.engelmann@uni-oldenburg.de

Dynamically typed languages enable programmers to write elegant, reusable and extendable programs. Recently, some progress has been made regarding their verification [2]. However, the user is currently required to manually show the absence of type errors, a tedious task usually involving large amounts of repetitive work.

As most dynamically typed programs only occasionally divert from what would also be possible in statically typed languages, properly designed type inference algorithms should be able to supply the missing type information in most cases [1].

We propose integrating a certified type inference algorithm into an interactive verification environment in order to
a) provide a layer of abstraction, allowing the users to verify their programs like in a statically typed language whenever possible and
b) use verification results to improve type inference and thus allow type checking of difficult cases.

As this is work in progress, we will in the following present the basic idea of our approach rather informally on the basis of small examples.

Model Language. For our investigation we use a minimalistic dynamically typed model language called *dyn* allowing us to focus on the problems arising from dynamic typing alone.

Dyn is imperative and object-oriented. It distinguishes syntactically between local variables (`v`) and instance variables (`@v`). The remaining syntax should be intuitively understandable.

Two built-in functions for checking object identity (`=`) and the runtime class of an object (`is_a`) are also provided. Primitives like numbers, strings and lists can be defined within *dyn*.

Type Inference. The goal of our type system is to statically prevent calling unsupported methods. As usual, the set of type error free programs is under-approximated as an exact solution is undecidable. We will later discuss some examples of correct programs that cannot be typed in our system.

Our type inference algorithm is based on [3], however the presentation focuses on what is necessary for the purpose of this writing. (* = Kleene star)

$$U ::= \{Classname^*\} \quad V ::= [\![S]\!] \quad T ::= V|U \quad C ::= T \subseteq T$$

The type language contains only union types (U) which denote all instances of the named classes. To infer these types for a given program P, we first assign each subexpression S a type variable $[\![S]\!]$ and then relate these variables with set inclusion constraints (C) generated by recursively traversing the parse tree of P.

This work is supported by the German Research Foundation through the Research Training Group (DFG GRK 1765) SCARE (www.scare.uni-oldenburg.de).

Oxhøj et al. [3] give a method for solving such constraint systems and properly handling dynamically dispatched method calls. In the end, we will either find the constraints to be inconsistent or have a proper solution σ assigning a union type $\sigma([\![S]\!])$ to every subexpression S of P.

Assertion Language. Our assertion language (P) contains the usual connectives from predicate logic and separation logic [4] along with typing assertions[2]. ($^+$ = Kleene plus)

$$\begin{array}{lll} P ::= & P \wedge P | P \vee P | \neg P | P \rightarrow P | (P) | \forall Id^+.P | \exists Id^+.P & T' ::= U | [\![E]\!] | [\![E.\texttt{@}Id]\!] \\ & | \texttt{emp} | P * P | P \mathrel{-\!\!*} P | E.@Id \mapsto E | E | T' \subseteq T' | T' = T' & \\ E ::= & \texttt{null} | Id | \texttt{@}Id | \texttt{self} | E.Id\texttt{(}E^*\texttt{)} | \texttt{new } Id\texttt{(}E^*\texttt{)} | \texttt{if } E \texttt{ then } E \texttt{ else } E \texttt{ end} | \texttt{result} & \end{array}$$

A subset of *dyn* expressions (E) can also be used in assertions[3]. The assertion language can thus be extended by the user as needed. However, method- and constructor calls are restricted to those proven to be total (terminate on all inputs) and side-effect-free. Contrary to programs, only well-typed expressions (inferable by our type inference algorithm) are allowed in assertions. In postconditions, `result` denotes the return value of the expression.

Examples. For most *dyn* programs (a polymorphic version of) the given type inference algorithm should be able to automatically infer the missing type information and thus allow reasoning like in a statically typed language. In the following, we will give some typical examples of non-typable programs and demonstrate how correctness can still be established.

Dynamic Type Check. In example ① the expression `self.howlong(3)` is not typable (class `numeric` does not support a method `length()`) although it can be executed without problems. The type system is not control flow sensitive and does not understand dynamic type checks (`is_a`). However, adding the assertion in line 3 corrects this problem. Assertions are control-flow sensitive as they denote properties of program states whose control flow reaches that point. The type expression $[\![\texttt{s}]\!]$ in the assertion thus denotes the type the parameter `s` has at that point rather than its general type. When encountering such an assertion, the constraint generation will introduce a new type variable $[\![\texttt{s}]\!]' = [\![\texttt{s}]\!] \cap \texttt{\{string\}}$ that is guaranteed to satisfy the given constraint and use it as the type of `s` until this assumption is invalidated (i.e. due to an assignment to `s` or a control flow join).

Note that such "type filters" need to be proven (like any other assertion) in order to preserve the type system's soundness. In this example, the proof is straightforward due to the postcondition of `is_a` and the semantics of the if statement.

Imprecise Control Flow Abstraction. In example ② we assume that numerics can be added to numerics and strings can be added (concatenated) with strings but there is no addition defined for combinations of the two (such attempts thus yield a type error). In this scenario, both `x` and `y` would be given the type `{numeric, string}` and example ② would not be typable as the algorithm would suspect that a `numeric` could be added to a `string` in line 10. Since the control flow is joined after the if statement, making the algorithm control flow sensitive does not help either.

Adding the assertion in line 9 solves this problem, since the expression `x + y` is well-typed under both possibilities. Establishing it is also simple as it holds at the end of both branches

[2] $[\![E.@Id]\!]$ denotes the type of the instance variable $@Id$ of the object referenced by E

[3] $E_1.@Id \mapsto E_2$ requires the instance variable $@Id$ of E_1 to point to E_2 and E is a shorthand for $E == true$

Examples:

```
1 method howlong(s) {
2   if s.is_a(string) then
3     #{⟦s⟧ ⊆ {string}}
4     s.length()
5   end                    ①
6 }
```

```
1  #{⟦l.@value⟧ ⊆ {string}}
2  method format(l) {
3    l.get(0) + ": " +
       l.get(1).to_string()
4  }
                ③
```

```
1  method do(b) {
2    if b then
3      x := "foo";
4      y := "bar"
5    else                   ②
6      x := 27;
7      y := 15
8    end
9    #{(⟦x⟧ ⊆ {numeric} ∧ ⟦y⟧ ⊆ {numeric})
         ∨(⟦x⟧ ⊆ {string}∧⟦y⟧ ⊆ {string})}
10   x + y
11 }
```

of the if statement due to the assignments and the standard rule for conditionals in Hoare logic states that a postcondition of both branches is also a postcondition of the entire statement.

Mixed-Type Container Elements. In dynamically typed languages, containers like lists are commonly used as records. Since type systems usually enforce all elements to be of the same type, such programs are not typable. In example ③, passing the list `["apples", 5]` to the method `format` would give `l.get(0)` the type `{string, numeric}` as instances of both classes are present in `l`. Consequently, the subsequent `+` operation yields a type error.

Now suppose the method `List.get(n)` had a postcondition `n == 0 → result = self.@value`. We could then conclude {`true`}`l.get(0)`{`result = l.@value`} by substitution and the rule of consequence. Since this implies ⟦`l.get(0)`⟧ = ⟦`result`⟧ = ⟦`l.@value`⟧, adding the precondition in line 1 allows us to conclude ⟦`l.get(0)`⟧ ⊆ `{string}`.

For lists starting with a string the precondition can easily be established and thus our program proven to be free of type errors.

Discussion. The sketched approach eases verification of dynamically typed languages by integrating type inference into the verification environment. We hope that the effort required for verifying dynamically typed programs will in many cases match their statically typed counterparts. For more complex typing problems the algorithm requires the user's help but should still be able to reduce the effort significantly.

References.

[1] Robert Cartwright and Mike Fagan. Soft typing. In *PLDI*, pages 278–292, 1991.

[2] Philippa Gardner, Sergio Maffeis, and Gareth David Smith. Towards a program logic for javascript. In *POPL*, pages 31–44, 2012.

[3] Nicholas Oxhøj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *ECOOP*, pages 329–349, 1992.

[4] Matthew J. Parkinson and Gavin M. Bierman. Separation logic for object-oriented programming. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming*, volume 7850 of *Lecture Notes in Computer Science*, pages 366–406. Springer, 2013.

Certified Normalization of Context-Free Grammars

Denis Firsov and Tarmo Uustalu

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

Abstract

Every context-free grammar can be transformed into an equivalent one in Chomsky normal form by a sequence of four transformations. In this work, we prove in the Agda programming language that each of these transformations is correct in the sense of making progress toward normality and preserving the language of the given grammar. Also, we show that the right sequence of these transformations leads to a grammar in Chomsky normal form (since each next transformation preserves the normality property established by the previous one) that accepts the same language as the given grammar. Since we work in a constructive setting, soundness and completeness proofs are functions converting between parse trees in the normalized and original grammars.

1 Introduction

In our previous work [2] we reported about a certified implementation of Cocke–Younger–Kasami (CYK) parsing algorithm in the Agda dependently typed programming language [1]. The CYK algorithm works only with grammars in Chomsky normal form. Now we extend the reach of this work by a certified implementation of the standard normalization transformation of general context-free grammars. This transformation is the composition of the following transformations:

1. eliminating all *ε-rules*;
2. eliminating all *unit rules*;
3. replacing all rules $N \to s_1 s_2 \dots s_k$ where $k \geq 3$ with rules $N \to s_1 N_1$, $N_1 \to s_2 N_2$, $N_{k-2} \to s_{k-1} s_k$ where N_i are new nonterminals;
4. for each terminal x adding a new rule $N \to x$ where N is a new nonterminal and replacing x in the right hand sides of all rules with N.

In this extended abstract, we present only the main definitions and describe the elimination of unit rules.

The full Agda code can be found at `http://cs.ioc.ee/~denis/cert-norm/`.

2 Setup

We assume that `NT` and `Tm` are some predefined types for nonterminals and terminals respectively and define a datatype for rules:

```
String = List Tm

data Symbol : Set where  % nonterminals and terminals
  nt : NT → Symbol
  tm : Tm → Symbol
```

```
RHS = List Symbol        % right-hand sides

data Rule : Set where
  _⟶_ : NT → RHS → Rule
```

For our purposes, it is sufficient to define a grammar as a list of rules:

```
Grammar = List Rule
```

The datatype of parse trees for a grammar is defined inductively as follows:

```
mutual
  data Tree (G : Grammar) : NT → String → Set where
    node : ∀ {L R xs} → (L ⟶ R) ∈ G
                      →  Forest G R xs → Tree G L xs

  data Forest (G : Grammar) : RHS → String → Set where
    empty  : Forest G [] []
    _::t_  : ∀ {R xs} → (x : T) →  Forest G R xs
                      → Forest G (tm x :: R) (x :: xs)
    _::n_  : ∀ {R xs N ys} → Tree G N ys → Forest G R xs
                      → Forest G (nt N :: R) (ys ++ xs)
```

The type `Tree G L xs` collects all parse trees for a string `xs` in the grammar `G` starting from a nonterminal `L`. The auxiliary type `Forest G R xs` collects all parse forests for a string `xs` whose constituent individual parse trees start with the symbols `R`.

3 Unit rules elimination and its correctness

We describe in list comprehension notation how unit rules with a particular right hand nonterminal are eliminated:

```
nur : Grammar → NT → Grammar
nur G N = [ rule' | rule ← G, rule' ← nur-f G N rule ]
  where
    nur-f : Grammar → NT → Rule → Grammar
    nur-f G N (L ⟶ R) =
       if R == [ nt N ] then
          [ L ⟶ R' | (L' ⟶ R') ← G, L' == N, R' != [ nt N ] ]
       else [ L ⟶ R ]
```

The function `nur G N` replaces rules of the form `L → [ nt N ]` with rules `L ⟶ R'`, where `R'` stands for right hand sides such that `(N → R') ∈ G`. Now full unit rules elimination is defined as

```
nur-full : Grammar → Grammar
nur-full G = foldl nur G NTs
```

where `NTs` is a list of all nonterminals in the grammar.

Progress First, we show that `nur` gains some progress towards normality:

```
nur-progress : ∀ {G L N} → (L ⟶ [ nt N ]) ∉ nur G N
```

This lemma states that there is no rule with right hand side `[ nt N ]` in the grammar `nur G N`. The similar progress lemma for `nur-full` is a consequence.

Soundness Soundness states that the language of the transformed grammar is a subset of the language of the original grammar.

We start by proving a lemma about possible shapes of rules in the original grammar:

```
nur-sound-main : ∀ {G N L R} → (L ⟶ R) ∈ nur G N
      → (L ⟶ R) ∈ G ∨ (L ⟶ [ nt N ]) ∈ G × (N ⟶ R) ∈ G
```

This lemma shows that, if a rule `L ⟶ R` belongs to grammar `nur G N`, then either the rule `L ⟶ R` belongs to `G` or the rules `L ⟶ [ nt N ]` and `N ⟶ R` do.

Now, soundness can be proved by applying lemma `nur-sound-main` to each level of a given parse tree.

```
nur-soundness : ∀ {G N S xs} → Tree (nur G N) S xs → Tree G S xs
```

Completeness Completeness states that the language of the original grammar is a subset of the language of the transformed grammar.

Again we start by proving a special property:

```
nur-complete-main : ∀ {G N L R} → (L ⟶ [ nt N ]) ∈ G → (N ⟶ R) ∈ G
      → R ≠ [ nt N ] → (L ⟶ R) ∈ nur G N
```

This lemma states that, if rules `L ⟶ [ nt N ]` and `N ⟶ R` belong to the grammar `G`, then the rule `A ⟶ xs` belongs to the transformed grammar `nur G N`.

Using this property, completeness is proved by induction on a given parse tree and analyzing rules at two consecutive levels and applying lemma `nur-complete-main`.

```
nur-completeness : ∀ {G N S xs} → Tree G S xs → Tree (nur G N) S xs
```

4 Conclusion

We have done this work in the constructive setting of the Agda programming language. Hence the soundness and completeness theorems are functions for conversion of parse trees between the normalized and original grammars. We have therefore attained our initial goal of extending certified CYK parsing to grammars in general form.

Acknowledgement This research was supported by the ERDF funded Estonian ICT national programme project "Coinduction", the Estonian Science Foundation grant No. 9475 and the Estonian Ministry of Education and Research target-financed research theme No. 0140007s12.

References

[1] A. Bove, P. Dybjer, U. Norell. A brief overview of Agda, a functional language with dependent types. In *Proc. of TPHOLs 2009*, v. 5674 of *LNCS*, pp. 73–78. Springer, 2009.

[2] D. Firsov, T. Uustalu. Certified CYK parsing of context-free languages. In *Abstracts of NWPT 2012*, v. 403 of *Reports in Informatics*, 3 pp. U. of Bergen, 2012.

Extending Abstract Behavioral Specifications with Erlang-style Error Handling*

Georg Göri[1], Bernhard K. Aichernig[1],
Einar Broch Johnsen[2], Rudolf Schlatte[2] and Volker Stolz[2]

[1] University of Technology, Graz, Austria
goeri@student.tugraz.at,aichernig@ist.tugraz.at
[2] University of Oslo, Norway
{einarj,rudi,stolz}@ifi.uio.no

1 Introduction

Software modeling languages traditionally abstract from low-level concerns such as the reliability and speed of the deployment architecture, to obtain concise and focused models [6]. However, modern distributed and virtualized systems are increasingly resource-aware, network-aware, and adapt to dynamically changing infrastructure. These concerns need to be captured at the modeling stage, but modeling languages must balance the need for abstraction with the need to express and analyze a system's ability to adapt to its environment.

The abstract behavioral specification (ABS) language targets distributed object-oriented systems [5], but does not currently support fault-tolerant features such as adaptability to distribution failures. This work develops an Erlang execution backend for ABS and extends ABS with error handling capabilities à la Erlang. The resulting extension of ABS combines rollback to invariant states at the object-level with Erlang style process linking and supervision.

2 ABS and Erlang

ABS is a statically typed object-oriented modeling language targeting distributed systems [5]. ABS is based on asynchronous method calls between Concurrent Object Groups (cogs), akin to Actors and concurrent objects. An asynchronous method call creates a new process in the called object which may run in parallel with the continuation of the calling process. The reply from the asynchronous call is placed in a future, a single-assignment global variable that can be shared between objects. Futures support retrieval and checking the availability of a reply. Whereas execution in different cogs happens in parallel, the execution of processes inside a cog is strictly interleaved and controlled by means of cooperative scheduling; i.e., explicit suspension points in the code allow the active process to be suspended and another local process to be activated. Suspension may be conditional; e.g., it can depend on the status of a future. ABS supports a proof theory for concurrent systems by means of local reasoning, based on seeing objects as monitor-like maintainers of class invariants [4]. This proof theory requires that the invariants hold at locally quiescent states; i.e., whenever a process may suspend it must ensure the invariant and whenever a process is scheduled it can assume the invariant.

Erlang is a dynamically typed functional programming language. Concurrency is done by lightweight processes which asynchronously exchange messages but do not share state [1]. Distribution with location transparent message passing is an integral part of Erlang. In addition to a standard exception mechanism, Erlang provides *process linking* to handle distribution and

*Partially funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (http://www.envisage-project.eu).

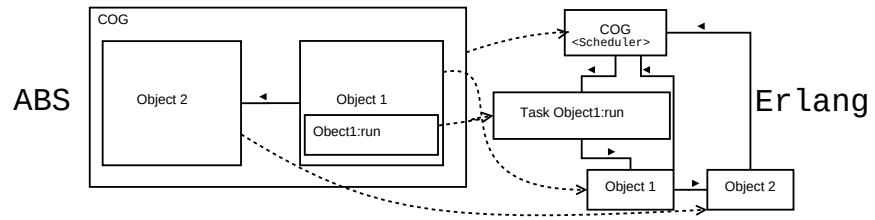


Figure 1: ABS entities mapped to Erlang processes, where solid lines show associations and dashed ones the mapping of components to processes.

runtime errors. A link is a bidirectional relationship between two processes, which guarantees the delivery of an exit signal to one process in case its partner terminates or becomes unreachable. These features allow *supervision* of processes, and enable a style of error handling where processes are allowed to crash and if needed, restart from a previous or initial state.

Our mapping of ABS to Erlang follows the principle that "everything is a process". While adhering to ABS semantics it provides distribution and scalability. Each cog becomes one Erlang process which controls local scheduling such that at most one ABS process can execute at a time. Each ABS process (and the main block) becomes one Erlang process which maintains the local variables. ABS objects become tail-recursive looping processes which handle field access via messages. Figure 1 depicts a runtime view of a model in both languages.

3 Error Handling in Distributed Models

This section presents an error handling schema for ABS. Instead of compensatory actions as in [2], we propose a system that combines error propagation via futures and automatic object-level rollback on failure. This way process linking and recovery operations can be implemented in ABS. Both runtime errors (e.g. division by zero, out of memory) and distribution errors (e.g. connection loss) are represented in the model. We introduce the following language constructs into ABS: a notion of user-defined error types; a generalization of the future mechanism to propagate either return values or errors; a statement **abort** e, which raises an error e and thereby terminates the process; a statement f**.safeget**, which can receive both errors and values from a future f; and a statement **die** which terminates the current object and all its processes.

To enable *error propagation*, ABS futures are enhanced to carry either the normal result of a method invocation or an error. The caller can retrieve the return value with the **get** expression, which will, in case the future carries an error e, lead to an implicit execution of **abort** e. If error handling is desired, the newly introduced **safeget** expression can be used, which will return the result wrapped either as `Value`$(< value >)$ or `Error`(e) data constructors. Presented design incorporates the Erlang principles: error propagation with fast failing as default or optional error recovery. The effect of executing **abort** is defined in the following way:

- **Active Object.** If the active object's process evaluates **abort** e, all processes of the object will abort with the error e and the references to this object will become invalid. Further synchronous or asynchronous calls are equivalent to executing **abort** `DeadObject`.
- **Asynchronous Call.** An **abort** e statement terminates the process, stores e in the associated future, and rolls back the object state. This rollback discards all changes since the last scheduling point and thus re-establishes the object invariant.
- **Main Block.** An **abort** here will not be further handled and the execution will terminate.

Towards linking of objects. The addition of a **die** statement, which has an equal effect to an **abort** in an active object (see above), enables us to implement a linking between Objects,

```
class Link(Linkable f,Linkable s){
  Int done=0;
  Unit setup(){
   f!waiton(this,s);
   s!waiton(this,f);
   await done==2;}
  Unit done(){
   done=done+1;}}
```

```
class Linkable() implements Linkable{
  Unit waitOn(Link l,Linkable la){
    Fut<Unit> fut=la!wait();
    l!done();
    await fut?;
    case fut.safeget {
     Error(e) => die e;
}}}
```

Figure 2: Implementation of Links in ABS

which is shown in Fig. 2. Error recovery code can replace the **die** statement in the `Linkable`. Support of the runtime system would drop the need to implement the `Linkable` interface in every class and could call automatically an optional error handling function.

4 Discussion

This paper reports on work connecting ABS to Erlang. The presented Erlang backend seems to scale well to a large number of ABS processes, and can be seen as a step toward a distributed implementation of ABS. This backend forms a basis for adapting Erlang's error handling capabilities to the statically typed object-oriented world of ABS. This paper extends ABS for such error handling; we are currently adapting the rewriting logic semantics of ABS. Next, we will apply the analysis tools of ABS to analyze error handling during the system design.

Two strands of related work are verification systems for Erlang and error handling systems in software models. For example, Castro et al. describe their experience in verifying properties of supervisor trees using McErlang [3]. Although McErlang directly takes Erlang code as input, checking the supervisor needed special patching, due to the lack of time simulation. While in ABS it could stay untouched, because time can be used both in execution and verification. Previous work on errors in ABS models proposes a compensation mechanism inspired by web services; it would be interesting to see how this approach could be integrated with the work reported here. The proposed model-based error handling allows the generation of concise test models for fault-tolerant distributed systems which can be explored by test-driven simulations. An interesting extension to the rollback-mechanism is to relate it to transactions. Another line of future work is to adapt Erlang's "failure free" way of message passing, where communication errors are ignored in send/receive events, and instead handled by monitors: in our setting, sending to an invalid process will lead to an abort immediately.

References

[1] Armstrong, J. *Programming Erlang.* Pragmatic Bookshelf, 2007.

[2] Johnsen, E. B., Lanese, I., and Zavattaro, G. Fault in the future. In *Coordination Models and Languages*, *LNCS* 6721, pages 1–15. Springer, 2011.

[3] Castro, D., Gulias V. M., Benac Earle, C., Fredlund, L.-Å., and Rivas, S. A case study on verifying a supervisor component using McErlang. In *PROLE 2010*, *ENTCS* 271, pages 23–40, 2011.

[4] Chang Din, C., Dovland, J., Johnsen, E. B., and Owe, O. Observable behavior of distributed systems: Component reasoning for concurrent objects. *J. of Log. and Alg. Prog.*, 81:227–256, 2012.

[5] Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. ABS: A core language for abstract behavioral specification. In *FMCO*, *LNCS* 6957, pages 142–164. Springer, 2012.

[6] Kramer, J. Is abstraction the key to computing? *Comm. ACM*, 50(4):36–42, 2007.

Towards A Core Language for Separate Variability Modeling

Alexandru F. Iosif-Lazăr[1], Ina Schaefer[2] and Andrzej Wąsowski[1]

[1] IT University of Copenhagen {afla|wasowski}@itu.dk *
[2] Technische Universität Braunschweig i.schaefer@tu-braunschweig.de

Introduction. Model-driven development of software product lines (SPL) [13, 3] exploits rich models of systems to represent the product line architecture, or *base model*, and variability models to describe and derive specific *variants*. Separate variability modeling involves two aspects: (i) describing the system's configuration space through a *feature model* (a set of features that characterize the products in the family); and (ii) describing the set of transformations applied to the base model by the activation of each feature, which we refer to as *variation points*. This architecture of variability models enables automatic *variant derivation* for any given legal configuration. For example, the HATS Abstract Behavioral Modeling Language [2] is actually a set of smaller languages that model each aspect separately and interact to provide the full model.

Feature models [7] (or alternatives such as decision models [12]) have been extensively researched and formalized and they are now widely used to represent SPL configuration spaces. However, each existing variability modeling language has redesigned the representation of transformations to the base models, while the automatic execution of these transformations has been implemented in an *ad hoc* manner.

Examples of general variability modeling approaches (that are independent of the language in which the base model is developed) include the *Orthogonal Variability Model* (OVM) [10], *Delta Modeling* [5, 11] and CVL [4]. OVM is a simple language and a methodology for superimposing variability over any software development artifact without interfering into its contents. Delta Modeling is a methodology that expands existing languages with statically executable variation points selected from a somewhat stable set. CVL is an industrial attempt to create a generic language that facilitates separate variability modeling for models specified in any MOF-based language [8]. It also demonstrates great flexibility through a wide range of variation points.

Our objectives are (i) to understand the execution semantics used by the aforementioned approaches and determine the core features required by separate variability modeling languages, and (ii) to provide the formal specification of a language that could be used in the development of a trustworthy product derivation tool.

We are motivated by the fact that trustworthy product derivation is essential to the development of safety critical embedded systems in domains such as automotive or industrial automation [1, 6]. Industrial standards such as IEC 61508 mandate the use of state of the art tools and quality assurance techniques. So far, the industry certifies individual products, or even avoids introducing any variability into safety critical parts of the systems[1]. Our goal is to facilitate the development of such systems and to enable usable certification strategies for product line tools.

*Supported by ARTEMIS JU under grant agreement n° 295397 and by Danish Agency for Science, Technology and Innovation

[1] Personal communication with partners in ARTEMIS projects.

A compact language for separate variability. We propose an abstract semantics of a core language for separate variability modeling as expressive and versatile as CVL. We will describe the language in a series of steps. First, we will formalize our representation of SPL base models. Second, we will introduce the fragment substitution variation point and show its syntax and the execution semantics for a set of variation points. Finally, we will give an overview of how the feature model drives the variant derivation.

Our **base models** are multigraphs of attribute-less untyped objects connected by directed links. Both objects and links are discrete entities with identity and we write $\mathbb{O}$ and $\mathbb{L}$ to denote the infinite universes of objects and links, respectively. We use the functions $\mathsf{src}, \mathsf{tgt} : \mathbb{L} \to \mathbb{O}$ to indicate the endpoints of each link.

A base model Bm is a pair $(BmObj, BmLnk)$ where $BmObj \in \mathbb{O}$ is a finite set of objects and $BmLnk \in \mathbb{L}$ is a finite set of links. We say that Bm is *closed under links* (or, simply, *closed*), meaning that for each link $l \in BmLnk$ its endpoints are contained in the model, so $\mathsf{src}\, l, \mathsf{tgt}\, l \in BmObj$. Similarly, any fragment f is a pair $(fObj, fLnk)$.

The **fragment substitution** in CVL is a very expressive variation point, as most of the other variation points can be easily reduced to it. However, we found its syntax unnecessarily complex. We simplify it such that a *fragment substitution fs* is a triple (p, r, Bdg) where p is a placement fragment, r is a replacement fragment and Bdg is a set of links called a *binding*. It is the *only* variation point used in our language. Its execution removes the placement from the model and uses the binding links to embed the replacement in its stead. Given a base model and a set of fragment substitutions Fs, we define well-formedness constraints that help both to facilitate formalization and to eliminate confusion about the effect of their execution.

The **execution semantics** of our fragment substitutions are captured by seven inference rules which test properties of each object and link and decide if they must be part of the variant. The execution simply iterates over all objects and links and copies those for which the rules apply, while skipping those for which no rule applies. For example, one of the rules copies objects from replacement fragments that do not occur in placement fragments in the same time:

$$\frac{(\_, r, \_) \in Fs \quad o \in r\,Obj \setminus \bigcup_{(p,\_,\_)\in Fs} p\,Obj}{o \in [\![Fs, Bm]\!]\,Obj}\ (\text{R-OBJ})$$

Another rule that copies links from replacement fragments also checks that the source and target of each link are also copied. This helps proving that by starting from a *closed* base model and executing a set of well-formed fragment substitutions we will obtain a *closed* variant model.

$$\frac{(\_, r, \_) \in Fs \quad l \in r\,Lnk \setminus \bigcup_{(p,\_,\_)\in Fs} p\,Lnk \quad \mathsf{src}\, l, \mathsf{tgt}\, l \notin \bigcup_{(p,\_,\_)\in Fs} p\,Obj}{l \in [\![Fs, Bm]\!]\,Lnk}\ (\text{R-LNK})$$

A **feature model** is a hierarchical structure of features that imposes dependencies on the variation points. A configuration is a set of features that are present in a variant. Each of these features determines the execution of a variation point. There can even be features that are activated multiple times so that the variation point is executed for each activation. We deal with this unavoidable complexity by proposing a flattening semantics which copies the variation points in a flat model on which we simply apply the fragment substitution execution semantics.

Implementation, confluence and translation validation. By providing a formally defined language we facilitate the implementation of a verifiable tool. A copying semantics can be implemented in declarative rule-based model transformation languages more easily and it is easier to argue about it using theorem provers.

The execution of our semantics is **confluent**. While the CVL specification suggests an implementation by in-place transformations (which makes the transformation order critical) our rules always produce the desired result independently of the order in which they are applied. This offers a great deal of flexibility to the SPL designer and paves the way to new ideas on how to implement variant derivation tools.

A verifiable correct implementation of a variability modeling language is hard to obtain. As an alternative, having a semantics formalized as inference rules enables the verification of the derivation through **translation validation** [9].

This approach to validation is independent of the actual implementation. What it does require is: (i) a common semantic framework for both the input and the output—which we provide by representing all models and fragments as graphs; (ii) a formalization of the notion of *correct execution*—which we provide in the form of inference rules and (iii) a proof method which allows to automatically verify that the output is correct.

Presentation. In the presentation we will discuss the main problem of variability modeling. We will introduce the formal semantics of our core variability language and give examples of graph manipulation via execution of fragment substitutions. We will also demonstrate the expressive power of fragment substitutions and how model variability specified in other languages can be rewritten using our syntax on a working example.

References

[1] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In S. Gnesi, P. Collet, and K. Schmid, editors, *VaMoS*, page 7. ACM, 2013.

[2] 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, *SFM*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.

[3] P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[4] CVL Joint Submission Team. *Common Variability Language (CVL). OMG Revised Submission*, 2012.

[5] A. Haber, K. Hölldobler, C. Kolassa, M. Look, K. Müller, B. Rumpe, and I. Schaefer. Engineering Delta Modeling Languages. In *SPLC*, 2013.

[6] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In R. N. Taylor, H. Gall, and N. Medvidovic, editors, *ICSE*, pages 633–642. ACM, 2011.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU SEI, November 1990.

[8] Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[9] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[10] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Sprinter Verlag, 2005.

[11] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC'10*, pages 77–91. Springer-Verlag, 2010.

[12] K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *VaMoS*, pages 119–126, 2011.

[13] T. Stahl and M. Voelter. *Model-Driven Software Development*. John Wiley & Sons, 2004.

A Categorical Foundation of Functional Reactive Programming with Mutable State

Wolfgang Jeltsch

TTÜ Küberneetika Instituut, Tallinn, Estonia
wolfgang@cs.ioc.ee

Abstract

Ordinary functional programming deals with values, which can be duplicated and discarded at will. Linear functional programming, on the other hand, deals with state, which can neither be duplicated nor discarded. Both paradigms can be combined within a single language, and their interaction can be modeled by a lax symmetric monoidal adjunction (LSMA).

In this paper, we show that a similar situation holds regarding functional reactive programming (FRP): we can combine ordinary functional programming and FRP and model their interaction by an adjunction that interacts with a cartesian endofunctor. Based on this observation, we develop a categorical structure that models the interaction of functional programming with linear functional programming and FRP at the same time. This structure enables us to obtain categorical models of an FRP variant with mutable state as pushouts in a suitable category.

1 Functional Reactive Programming

Functional reactive programming (FRP) is a programming approach that deals with temporal aspects in a declarative way. It is the Curry–Howard correspondent of an intuitionistic temporal logic.

We have developed several kinds of categorical models of FRP in our earlier work [2, 3, 4]. These assume that time is linear, but put no further restrictions on the time scale. In this paper, we additionally assume that time is discrete. This enables us to use a much simpler categorical semantics, which we describe in the remainder of this section.

A categorical model of FRP contains a cartesian closed category (CCC), which we call $\mathcal{T}$ here. The objects of $\mathcal{T}$ model FRP types. Type inhabitance in FRP is time-dependent. If objects A and B model FRP types τ_1 and τ_2, a morphism $f : A \to B$ denotes an operation that turns any value that inhabits τ_1 at some time into a value that inhabits τ_2 at the same time. Finite products and exponentials in $\mathcal{T}$ model product types and function types, respectively.

Besides the CCC $\mathcal{T}$, a categorical model of FRP contains a cartesian endofunctor $\bigcirc$. If an object A models a type τ, $\bigcirc A$ models a type whose inhabitants at a time t are the inhabitants of τ at time $t + 1$.

2 Linear Functional Programming

Linear functional programming (LFP) does not deal with ordinary values but with the state of mutable objects. It uses a linear type system, which prevents state from being duplicated or discarded. LFP is the Curry–Howard correspondent of intuitionistic linear propositional logic.

We can model LFP by a symmetric monoidal closed category (SMCC). In an SMCC $(\mathcal{L}, \otimes, I, \multimap)$, the tensor $\otimes$ and the identity I model the type constructors for binary and nullary tuples, and the right adjoint $\multimap$ of $\otimes$ models the function type constructor.

Ordinary functional programming and LFP can be combined and the resulting system can be given a categorical semantics based on lax symmetric monoidal adjunctions (LSMAs) [1]. A categorical model according to this semantics consists of the following components:

- a CCC $\mathcal{C}$, which models the non-linear part
- an SMCC $(\mathcal{L}, \otimes, I, \multimap)$, which models the linear part
- an LSMA $(F, \varphi, \psi) \dashv (G, \upsilon, \nu)$ between $(\mathcal{C}, \times, 1)$ and $(\mathcal{L}, \otimes, I)$, which models the interaction between the two parts

The functor F models a type constructor whose inhabitants are values treated as state, while the functor G models a type constructor whose inhabitants are values that describe the generation of mutable objects.

Note the general idea behind this approach of modeling functional programming, LFP, and their interaction:

- We model both functional programming and LFP by a category with additional structure (an SMCC structure).
- For the model of functional programming, we require that the additional structure is of a specific kind (a CCC is a specific kind of SMCC).
- We model the interaction between functional programming and LFP by an adjunction that interacts with the additional structure of the two categories (by being an LSMA).

3 Interaction between Functional Programming and FRP

We can combine functional programming and FRP in a way analogous to combining functional programming and LFP. Let $(\mathcal{T}, \bigcirc)$ be a categorical model of FRP. Its "additional structure" is the cartesian endofunctor $\bigcirc$. We take a category $\mathcal{C}$ for modeling functional programming and give it a cartesian endofunctor $\bigcirc$ as well. However we choose a specific cartesian endofunctor structure: we define $\bigcirc$ to be the identity functor.

We model the interaction between functional programming and FRP by an adjunction $F \dashv G$. The functor F models a constructor of FRP types whose inhabitants are those of an ordinary type, independently of time. The functor G models a constructor of ordinary types whose inhabitants are those that inhabit an FRP type at every time.

We let the adjunction $F \dashv G$ interact with the cartesian endofunctors by requiring that there exist natural transformations of types $F\bigcirc \to \bigcirc F$ and $\bigcirc G \to G\bigcirc$ that fulfill obvious axioms. Since $\bigcirc = \mathrm{Id}$ for the category $\mathcal{C}$, this boils down to having natural transformations of types $F \to \bigcirc F$ and $G \to G\bigcirc$. The first of them models an operation that stores a value of an ordinary type until the next time; the second of them establishes that time-universal values are valid at every time $t + 1$.

4 FRP with Mutable State

Now we integrate the structures of Sects. 2 and 3:

- We model each of the paradigms functional programming, LFP, and FRP by an SMCC with a symmetric monoidal endofunctor (SME), which we name $\bigcirc$. We call the underlying categories $\mathcal{C}$, $\mathcal{L}$, and $\mathcal{T}$.
- We require that the SMCC structures of $\mathcal{C}$ and $\mathcal{T}$ are CCC structures, and that the functor $\bigcirc$ of $\mathcal{C}$ and $\mathcal{L}$ is the identity functor.
- We require the existence of two LSMAs that interact with $\bigcirc$: one between $(\mathcal{C}, \times, 1, \Rightarrow, \mathrm{Id})$ and $(\mathcal{L}, \otimes, I, \multimap, \mathrm{Id})$, another between $(\mathcal{C}, \times, 1, \Rightarrow, \mathrm{Id})$ and $(\mathcal{T}, \times, 1, \Rightarrow, \bigcirc)$.

An implication of the above requirements is that the adjunction between $\mathcal{C}$ and $\mathcal{T}$ preserves products, something that we did not enforce before.

Now consider the category whose objects are SMCCs with an SME, and whose morphisms are LSMAs that interact with the SMEs through appropriate natural transformations. The constructs described above form a span in this category. A pushout of this span models a linear form of FRP, that is, an FRP variant that can deal with mutable state.

Note that there is an LSMA between $(\mathcal{T}, \times, 1, \Rightarrow, \bigcirc)$ and the pushout that has two associated natural transformations of types $F\bigcirc \to \bigcirc F$ and $\bigcirc G \to G\bigcirc$. We assume that an object $\bigcirc A$ of the pushout models a type whose inhabitants generally do not just denote future state, but may denote effectful computations that take one time step to produce a state. Note that even under this assumption, the types of the abovementioned two natural transformations make sense:

1. A future value (treated as a state) can be turned into a computation that delivers this value in the future (again treated as a state). This computation is special in that it does not have any actual effect.
2. A future generator of mutable objects can be turned into a generator of computations that deliver mutable objects in the future. The generated computations just wait until the next time and then run the original generator. They are also special in that they do not have any actual effect.

Acknowledgement This research was supported by the ERDF funded Estonian ICT national programme project "Coinduction" and the Estonian Ministry of Education and Research target-financed research theme No. 0140007s12.

References

[1] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. Technical Report UCAM-CL-TR-352, University of Cambridge, Cambridge, England, October 1994.

[2] Wolfgang Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science*, 286:229–242, September 2012.

[3] Wolfgang Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*, pages 69–78, New York, 2013. ACM.

[4] Wolfgang Jeltsch. An abstract categorical semantics for functional reactive programming with processes. Accepted for PLPV '14, January 2014.

On Exploiting Progress for Memory-Efficient Verification of Diagrammatic Workflows

Lars Michael Kristensen[1], Yngve Lamo[1], Wendy MacCaull[3], Fazle Rabbi[1] and Adrian Rutle[2*]

[1] Bergen University College, Bergen, Norway
Lars.Michael.Kristensen@hib.no, Yngve.Lamo@hib.no, fra@hib.no
[2] Alesund University College, Alesund, Norway
adru@hials.no
[3] St. Francis Xavier University, Antigonish, Canada
wmaccaul@stfx.ca

1. Introduction: Workflow management tools may be used in many domains, to guide and direct processes, to support monitoring activities and to increase organizational efficiency. Clinical practice guidelines are textual guidelines describing treatments for specific health problems. Problems can arise if the guideline is misunderstood by the user, if the guideline itself is incomplete, inconsistent or ambiguous, or, if two more guidelines are being followed simultaneously for a patient with several problems. In safety critical applications such as healthcare, it is essential that the workflow is error-free, that is, for every execution of the workflow, necessary behavioural requirements are satisfied and unwanted behaviours do not occur. In earlier work [5], we have proposed a model-driven engineering (MDE) based approach to workflow modelling, with the goals to provide a framework that can model typical healthcare protocols, by means of a visual tool which can be easily understood by the users (usually clinicians), and to articulate and model check behavioural properties. With this approach, the user can input a workflow model and workflow properties which are defined diagrammatically; the model is automatically transformed to DVE code (the DiVinE model checker's language) and the properties to LTL-formulae [6]. If the workflow model is not valid wrt. a property, the tool provides a visual representation of a path which is a counter-example that can be easily analysed for debugging purposes. The verification technique applied is based on explicit state space exploration where all states and all execution paths are explored to check whether the property holds. Most workflow models are so complex that their analysis lead to state-space explosion and memory overflow. In this paper, we propose to use the sweep-line method to exploit inherent progress present in health-care workflows to combat the state explosion problem.

2. Background: Workflow modelling: The syntax and semantics of the workflow modelling language which we use in our approach may be found in [5, 6]. The modelling language is defined using the Diagram Predicate Framework (DPF) [4] and implemented using the DPF Workbench [3]. In DPF, a modelling language is given by a metamodel and a diagrammatic predicate signature (see Fig. 1). The metamodel defines the types and the signature defines the predicates that are used to formulate constraints. A metamodel in DPF consists of an underlying graph, and a set of constraints. We say that a model conforms to (is an instance of) a metamodel if the model's underlying graph is typed by the metamodel's underlying graph, and if the model satisfies the constraints defined in the metamodel. In DPF, the semantics of a (meta)model is given by the set of its instances. DPF supports a multi-level metamodelling hierarchy, in which a model at any level can be regarded the metamodel for models at the level below it.

*in alphabetical order by last name

In the design of our workflow modelling language we have three modelling levels: M2, M1 and M0 (see Fig. 1). The metamodel of our workflow modelling language (which is at level M2) consists of a node Task and an arrow Flow. Simply put, this means that we can define a set of tasks together with the flow relations between these tasks. The signature Σ_2 of the workflow modelling language consists of a set of routing predicates such as `[and_split]`, `[and_merge]`, `[xor_split]`, and `[xor_merge]`. In addition, `[NodeMult,n]` is used to restrict the number of instances (n) a task can have; i.e., it controls how many times a task can be performed, and is used as an upper bound in loops (cycles) in workflow models.

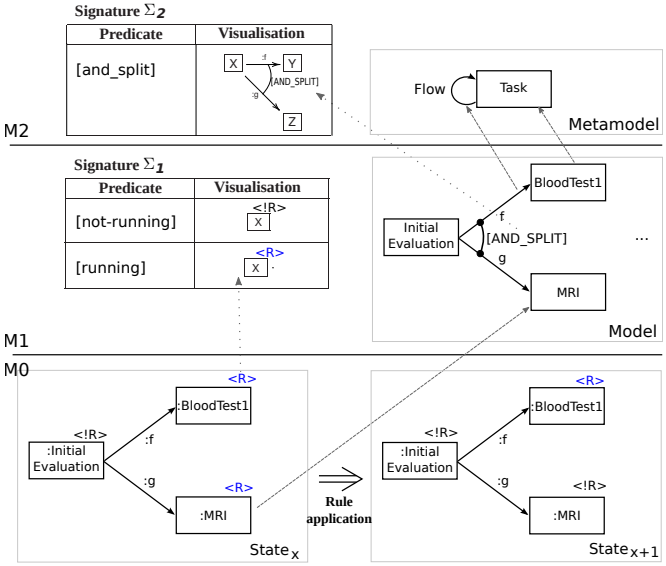


Figure 1: Workflow modelling hierarchy: the dashed arrows indicate the types of model elements, the dotted arrows indicate the relation between the signatures and the models

Given a specific workflow model at level M1 (like the one in Fig. 1) and the predicates `[running]` and `[not-running]` (denoting, respectively, that a task instance is running or not running) collected in a signature Σ_1, we create another modelling language which we use to define *workflow states*, or, equivalently, *instance of a workflow model*. The workflow states are located at level M0 we generate states by applying model transformation rules.

Sweep-line method: The sweep-line method is a state space reduction technique based on the paradigm of on-the-fly state deletion [2]. In order to determine the subsets of states that are to be stored in memory, the sweep-line method exploits a notion of progress exhibited by many systems. The method explores all reachable states while storing only small subsets of the state space in memory at a time. The basic idea of the sweep-line method is when it observes states with a higher progress value, it deletes the states with a lower progress value. It optimistically assumes that the system does not regress; if it turns out that the system does regress to a state with a lower progress value, the method will mark those states as *persistent* (i.e., make them permanently stored in memory) in order to ensure termination [2]. Formally, the progress in a system is formalised by a progress mapping that maps each state into a progress value, and a total order on progress values which is used to determine when states can be deleted.

3. Initial Workflow Verification with the Sweep-Line Method: We performed some verification experiments with workflow models with a small number of tasks. Fig. 2 illustrates a simplified scenario for a cancer treatment. After an initial examination, the patient has an MRI examination and a blood test. These tasks are performed concurrently. An evaluation of the results of the two tests is performed when both tests are completed so the physician can determine which procedure the patient should follow (*either* ProcedureA *or* ProcedureB).

After finishing this procedure, a second evaluation occurs to determine if the patient should continue with a drug treatment or if this workflow should end. The tasks Evaluation2, TakeDrug and BloodTest2 will be repeated up to 5 times, indicated by the loop counter. To specify a

progress measure for a state in the current sweepline implementation for workflow verification, we considered the ordering of tasks; $P_1, P_2, ..$ in Fig. 2 shows the positions of tasks ordering. The ordering shown in the figure is a total order that we obtained from the structural partial ordering of tasks (e.g., $(P_1 < P_2 < P_4)$, $(P_1 < P_3 < P_4)$, $(P_4 < P_5 < P_9)$, etc,.). Clearly, it is also possible to have other total ordering such as $P_1, P_3, P_2, P_4...$ for the given workflow model.

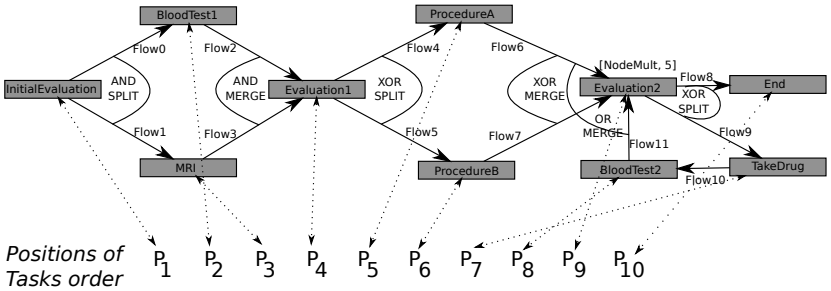


Figure 2: Cancer treatment workflow model

For a specific total ordering if we use $0/1$ in $P_1, P_2, ..$ to represent *not-running/running* task states and an integer number for the loop counter (0-5 for P_9) then the reverse of the number (i.e., right to left ordering) represents the progress value for a workflow state. In situations when we iterate on a loop in our workflow (i.e., Evaluation2 task), the sweep-line method makes some states persistent during the second sweep iteration.

For the cancer treatment workflow model we could define a non-monotonic progress measure by exploiting the loop counter (e.g., the value of P_9 increases from 0 to 5) in the progress measure. If we do not exploit the loop counter (that means if we use $0/1$ for P_9) then we get a monotonic progress measure; but of course still explore the entire state space (albeit some states may be visited multiple times). For the cancer treatment workflow model in Fig. 2, we obtained a reduction of 85% in peak memory usage with the sweep-line method.

In [1] authors presented a sweep-line algorithm for Buchi automata based model checking. The algorithm can detect accepting cycles and can reduce states on the fly; therefore sweep-line methods can be used efficiently for the verification of LTL properties. In future we will evaluate the sweep-line method for the verification of large workflow models and will investigate: i) further reduction using external memory, ii) automated discovery of progress measure, iii) investigate the time/space trade-off when using non-monotonic progress measures.

References

[1] S. Evangelista and L. M. Kristensen. A sweep-line methods for buchi automata-based model checking. *In Fundamenta Informaticae, 2013, to appear.*

[2] K. Jensen, L. M. Kristensen, and T. Mailund. The sweep-line state space exploration method. *Theor. Comput. Sci.*, 429:169–179, 2012.

[3] Y. Lamo, X. Wang, F. Mantz, W. MacCaull, and A. Rutle. DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment. In *Computer and Information Science*, volume 429 of *Studies in Computational Intelligence*, pages 37–52. Springer, 2012.

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

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

[6] A. Rutle, F. Rabbi, W. MacCaull, and Y. Lamo. A user–friendly tool for model checking healthcare workflows. *The 3rd International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2013), to appear.*

The Advantage of Using Co-span Graph Transformations for Meta-model Evolution*

Florian Mantz[1] and Uwe Wolter[2]

[1] Bergen University College
fma@hib.no
[2] University of Bergen
uwe.wolter@ii.uib.no

Model-driven engineering [5] (MDE) is a software engineering discipline which employs models as primary artifacts in the software development process. Software (parts) are specified using modeling languages at an high abstraction level. Model-transformations are used to automate recurring development tasks as well as to generate software artifacts for different runtime environments and testing. This can improve productivity of developers as well as quality and cost-effectiveness of software. However, to truly gain benefits from MDE it is important that used modeling languages suit well to their modeling purposes. Therefore, domain-specific modeling languages designed for specific tasks that are continuously improved have become popular. However, the evolution of modeling languages introduce a new challenge developers struggle in practice with, existing models need to be migrated after the corresponding modeling language has been evolved (see. Fig. 1). Since the manual migration of models is tedious and error-prone, different approaches have been developed that (partially) automate this task.

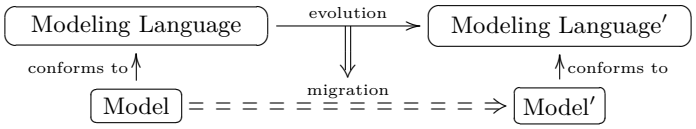


Figure 1: Model co-evolution: Modeling language evolution and model migration

In our work, we focus on the formalization of this co-evolution challenge using category theory [1]. In particular, we use algebraic graph transformations [3] to describe modeling language evolution and model migration as sequences of rule applications. Modeling languages are specified in meta-models. In this paper, we focus on the meta-model evolution task and refer to our earlier work [8, 7, 10] for the model-migration task. In algebraic graph transformation, it is prevalent to describe transformation rules as spans, while it is also possible to use co-spans [4, 6]. In our formalization we are using co-spans and in this paper we argue why co-span rules suit better to the meta-model evolution task. Additional reasons why co-spans rules also better fit to the model migration challenge can be found in [10]. Figure 2 shows a graph transformation using a span rule while Fig. 3 shows a graph transformation using a co-span rule. A graph transformation is applied by constructing either two pushouts (double pushout approach[3]) or one pushout and one pullback (sesqui pushout approach [2]).

In the following, we will explain why co-span rules suit better to the meta-model evolution challenge. In MDE, model migration approaches can be categorized into three different kind of approaches [9]:

1. *Manual specification approaches:* model migrations have to be specified manually while there is some support to migrate unchanged model parts automatically.

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

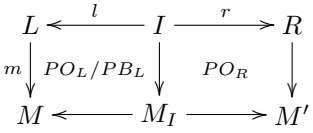


Figure 2: Graph transformation span approach

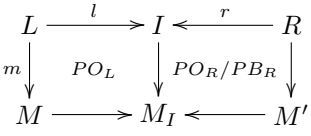


Figure 3: Graph transformation co-span approach

2. *Operator based approaches:* coupled evolution/migration operators are used to evolve the meta-model and migrate models correspondingly.
3. *Matching approaches:* a sequence of meta-model evolution steps is detected automatically and models are migrated correspondingly.

While the first kind of approaches (manual specification) are uninteresting for this paper, describing meta-model evolution as a sequence of rule applications fits obviously well to the second kind of approaches (operator based). However, using co-span rules instead of span rules to describe meta-model evolution steps helps also to formalize the third kind of approaches (matching) as we explain next. Therefore consider Fig. 4 and Fig. 5 first: both figures show a meta-model evolution of a meta-model M_1 to M_7 described by a sequence of three rule-applications using span rules in Fig. 4, respectively co-span rules in Fig. 5. In both cases, the first (M_1) and last meta-model (M_7) can be related by a span respectively co-span by composing pullbacks, respectively pushouts as shown in the example figures.

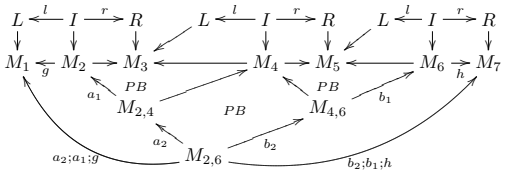


Figure 4: Meta-model span composition

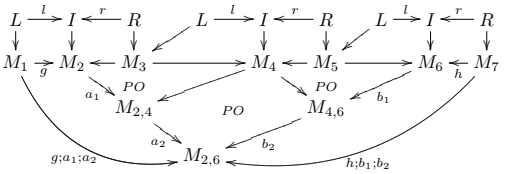


Figure 5: Meta-model co-span composition

Figure 6 shows a concrete example for meta-model composition with co-spans. A sequence of three rules "Move Attribute", "Merge Class", "Add Container" has been applied to a simple meta-model. Only the meta-models are shown and their compositions by pushouts. By composing morphisms meta-model M_1 and M_7 can be related by co-span $M_1 \to M_{2,6} \leftarrow M_7$. Looking at this meta-model co-span it seems to be feasible that a tool could create it without knowing any possible rule application sequence. A tool can analyze the ids of the meta-model elements instead. (We assume that ids of meta-model elements do not change and also that merged elements can be identified by the corresponding set of ids.)

Having a meta-model co-span, the co-span can be incrementally decomposed using the following procedure:

Procedure 1 (Decompose meta-model co-span)**.** *(see Fig. 7)*
Given meta-model co-span $M_1 \overset{a_{max}}{\to} M_{max} \overset{b_{max}}{\leftarrow} M_N$ *and a set of co-span rules* R*.*

1. *Find a triple match* $<m_L, m_I, m_R>$ *of a co-span rule* $r \in R$ *in* $M_1 \to M_{max} \leftarrow M_N$*.*
2. *Apply co-span rule* r *to meta-model* M_1*.*
3. *Obtain* $d : M_2 \to M_{max}$ *as mediating morphism from the left pushout* PO_L *of* r*'s rule application.*

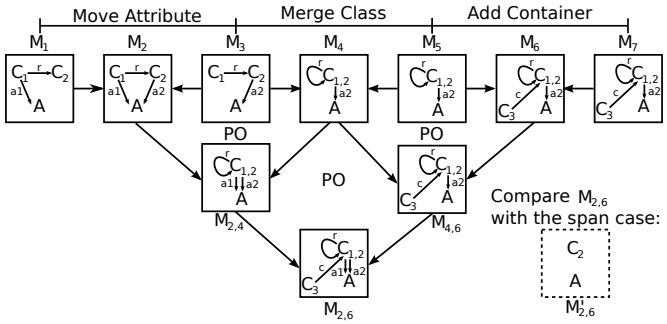


Figure 6: Example: Meta-model co-span composition

4. *Construct* $M_3 \leftarrow M_{sub} \rightarrow M_N$ *as pullback of* $M_3 \overset{h;d}{\rightarrow} M_{max} \overset{b_{max}}{\leftarrow} M_N$.
5. *Construct new meta-model co-span* $M_3 \overset{a_{max-1}}{\rightarrow} M_{max} \overset{b_{max-1}}{\leftarrow} M_N$ *as pushout of* $M_3 \leftarrow M_{sub} \rightarrow M_N$.
6. *Continue to decompose co-span* $M_3 \overset{a_{max-1}}{\rightarrow} M_{max-1} \overset{b_{max-1}}{\leftarrow} M_N$ *(recursion).*
   Stop if $a_{max-1} = id = b_{max-1}$ *or no more triple matches found.*

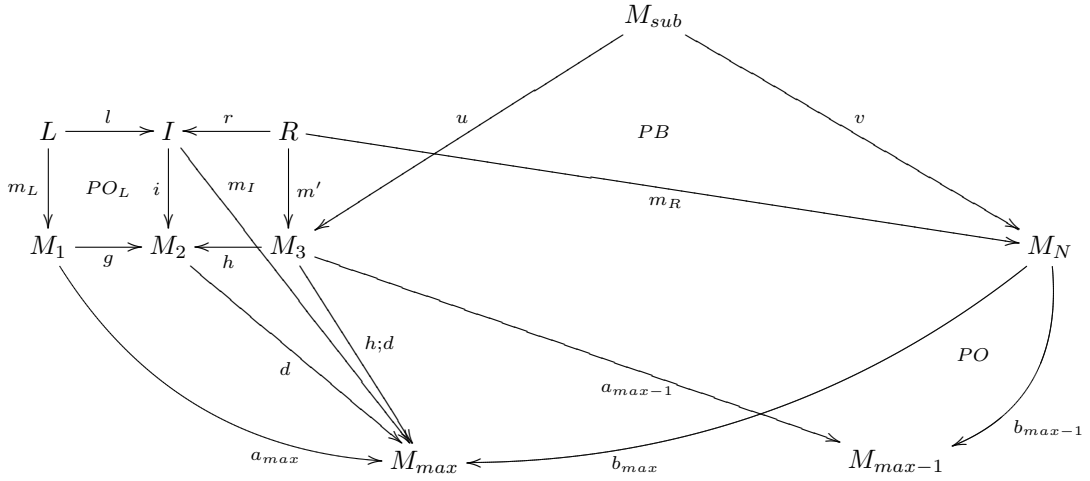


Figure 7: Meta-model co-span decomposition

Remark 1.

1. *Rule matches can be searched in parallel.*
2. *Procedure 1 is also applicable if the sesqui pushout approach is used. (The pushout property of the second rule application is not used in the procedure.)*
3. *If non-injective rules are used, non-injective matching may be required.*

Figure 8 shows an example decomposition step. In each further decomposition step the difference between M_{max-1} and M_{sub} becomes smaller. A particular sequence is not guaranteed.

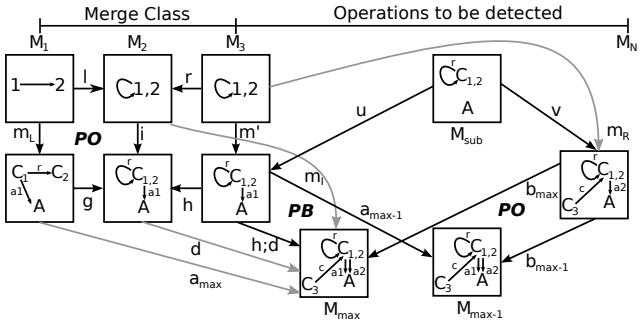


Figure 8: Example: Meta-model co-span decomposition

Having a meta-model span relation such a decomposition is hardly feasible; in Fig. 6 the dashed part shows the intermediate meta-model in case we would have used a meta-model span. Hence, a co-span approach seem to fit better to the meta-model evolution challenge and in particular to the formalization of matching approaches.

References

[1] M. Barr and C. Wells. *Category Theory for Computing Science (3^{rd} Edition*. Les Publications CRM, Montreal, 1999.

[2] A. Corradini, T. Heindel, F. Hermann, and B. König. Sesqui-Pushout Rewriting. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *ICGT 2006*, volume 4178 of *LNCS*, pages 30–45. Springer, September 2006.

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

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

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

[6] Y. Lamo, F. Mantz, A. Rutle, and J. de Lara. A declarative and bidirectional model transformation approach based on graph co-spans. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 1–12, New York, NY, USA, 2013. ACM.

[7] F. Mantz, G. Taentzer, and Y. Lamo. Co-Transformation of Type and Instance Graphs Supporting Merging of Types with Retyping. In *GCM 2012*, pages 47–58, September 2012. `http://gcm2012.imag.fr/proceedingsGCM2012.pdf`.

[8] F. Mantz, G. Taentzer, and Y. Lamo. Well-formed Model Co-evolution with Customizable Model Migration. *ECEASST*, page (accepted paper), March 2013.

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

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

Synchronization Property Checking and Inference in a Lock-Step Synchronous Parallel Replica Language

Jari-Matti Mäkelä[1], Ville Leppänen[1] and Martti Forsell[2]

[1] University of Turku
Dept. of Information Technology
jmjmak@utu.fi, ville.leppanen@utu.fi
[2] VTT
Oulu, Finland
Martti.Forsell@VTT.Fi

As more and more computer platforms adopt the model of multiple processing cores to get further speedup from parallelism, programming languages also need to adapt. Contemporary hardware solutions include heterogeneous systems with units optimized for different tasks, distributed memory clusters, and shared memory systems with different consistency models (e.g. on one end constrained platforms such as the GPGPU and on the other end more general purpose systems such as NUMA x86). As chip multi-processors provide new potential for organizing data sharing, we only consider languages targeting these systems.

We argue that in the case of general purpose algorithms, both the hardware and software solutions approach the issue with suboptimal abstractions, effectively preventing maximal utilization of the computational power. Parallelism is typically orchestrated with explicit locks, which leads to difficult problems with performance scaling and correctness [2]. Some platforms provide sequential consistency for safer programming with locks, but even more relaxed models [1] are in use in performance oriented languages such as C++.

We assume a totally different model for computation with strong synchronization guarantees. The SB-PRAM [6], TOTAL ECLIPSE [4], and REPLICA [5, 7] platforms conceptually follow the PRAM (*parallel random access memory*) model of computing. That is, each thread's computation proceeds in a globally synchronous lock-step at instruction level. Previously the model was considered too inefficient to be practical compared to more relaxed execution models. However, modern techniques of parallel *multi-operations, latency hiding* and *wave synchronization* alleviate the issues with the approach.

In this model, instructions have a unit time amortized cost in terms of cycles in relation to other threads, which makes it is possible to reason about the time cost of a sequence of instructions assuming no branching happens. This opens up a possibility for a different programming style where threads can be grouped so that the group property holds through a sequence of code and abstractions can make assumptions of the synchronicity of a group of threads. However, the hardware provided synchronicity on instruction level does not implicitly extend to higher level abstractions because the control flow structures may diverge the flow until explicitly synchronized with a barrier.

The *Fork* [6] language adopts a similar approach for maintaining synchronicity on block and function level, but does it explicitly with user annotated regions (*async, sync, straight*) that are statically checked. The major disadvantages of Fork's approach are 1) the need for tedious bookkeeping when switching between asynchronous and synchronous modes both at call site and in function signatures and 2) the limitation of reusing code in different synchronicity context. However, Fork does prevent erroneous use of synchronicity from compiling and provides a way to structure code into parallel and sequential sections.

Our approach originally presented in [8] attempts to automate the chore of tagging code with synchronization metadata like in Fork. In the previous work, each syntactic language entity (expression, statement, function, higher order abstraction) is associated with *synchronicity pre- and post-conditions* and intertwining constraints. The mechanism not only covers the case of *async/sync/straight* blocks of code like in Fork, but supports arbitrary conditions assuming the checking algorithm is provided with constraints involved additional conditions. As an example, we consider the synchronization token attribute that can be used to accelerate multi-operations on architectures such as REPLICA. New constraints could be provided as part of the compiled program, but in the case of Replica language, for simplicity we only use a static constraint set defined in the compiler.

A short summary of the checking algorithms is given next. For each syntactic language entity, the properties are represented by a pair of in- and outgoing sets of attributes ($\langle F_1, F_2 \rangle$) and the related constraint is a logical boolean predicate C representing a set invariant that covers both F_1 and F_2. On the implementation side, the compiler has an additional control flow analysis pass that carries the synchronicity condition state throughout the program in one pass and for each entity checks that the conditions are satisfied. A constraint rule is required for each language entity. The previous work also discussed so called "implicit conversion" rules between the states when transitioning between different sections with respect to synchronicity.

The problem of generating compatible conditions that satisfies the constraints for all parts of the program – that is, an inference algorithm – was not fully addressed in the previous work although it was mentioned that the constraint rules give rise to a similar inference technique as with e.g. standard typed lambda calculus. In the case of rules with no state dependencies, its state can be easily inferred, but there is no single solution in the case when states form complex dependencies between several syntactic nodes. While the condition checking against the constraints can be performed without backtracking or lookup in a linear fashion with respect to the syntax tree traversal, the inference algorithm requires arbitrary lookup for determining the need and position for code generation.

The main contribution of this work is to provide a way to automatically tag the test the correctness of the tagging is given for a minimal core of the Replica language. We propose a solution that handles basic functions and blocks with multiple exit points. We also revisit the idea of duality between rules for constraint checking and inference as discussed in the previous preliminary paper and elaborate how the mechanism extends to user defined abstractions.

In addition to the treatise of the inference and checking algorithm, we show the performance implications of the machine checked and generated code versus explicitly stated synchronization directives in computational kernels. The performance effect of the synchronization token is also measured with a task parallel runtime stub library that spawns a single task for evaluation purposes. Further applications for the synchronicity conditions in faster branching control flow constructs (so called "fast operations" in the E language [3]) are also considered. The goal of the benchmarks is to show the performance advantage of the approach with inferred synchronization properties versus a simpler, more generic algorithm that does not assume a strictly synchronous mode as a default nor does have hardware acceleration support for fast operations or special task tokens.

References

[1] Sarita V Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *computer*, 29(12):66–76, 1996.

[2] M. Duranton, D. Black-Schaffer, S. Yehia, and K. De Bosschere. Computing systems: Reseach challenges ahead – the hipeac vision 2011/2012, 2012.

[3] M. Forsell. E – A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs. *WSEAS Trans. on Computers*, 3(3):807–812, jul 2004.

[4] M. Forsell. TOTAL ECLIPSE – An Efficient Architectural Realization of The Parallel Random Access Machine. *Parallel and Distributed Comput., Ed. A. Ros, IN-TECH, Wien*, pages 39–64, 2010.

[5] M. Forsell. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. *Int. Journal of Networking and Computing*, 1(1):21–35, 2011.

[6] C.W. Kessler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of parallel programming*, 1997.

[7] J-M. Mäkelä, E. Hansson, M. Forsell, C. Kessler, and V. Leppänen. Design Principles of the Programming Language Replica for Hybrid PRAM-NUMA Many-Core Architectures. In *Proceedings of 4th Swedish Workshop on Multi-Core Computing*, page 136. Linköping University, 2011.

[8] J-M. Mäkelä, V. Leppänen, and M. Forsell. Composable Hierarchical Synchronization Support for REPLICA. In Kiss kos, editor, *13th Symposium on Programming Languages and Software Tools*, pages 230–244. University of Szeged, 2013.

Event Structures as Psi-calculus

Håkon Normann*

Dept. of Informatics – Univ. of Oslo, P.O. Box 1080 Blindern, 0316 Oslo. E-mail: haakno@ifi.uio.no

Abstract

Psi-calculi have been recently introduced as a generalization of π-calculi involving nominal data structures and powerful conditionals and assertions [BJPV11]. Instantiations of psi-calculi become standard variants of pi-calculus like the cryptographic, polyadic, or distributed extensions. We are interested in this paper in how psi-calculi could accommodate the event structures model of concurrency, with a final goal of capturing the Dynamic Condition Response graphs model (DCR-graphs). Event names in event-based models of concurrency are unique, and can thus be thought of nominals, whereas the execution of an event can be seen as a transmission of some sort. The dependencies between events that an event structure defines can be captured with rather simple assertions on the nominal data structures, and similarly the notion of computation in event structures.

These are the basic ideas the we follow in this work to give an encoding of event structures into an instance of psi-calculi. The drawback is that psi-calculi have interleaving semantics using rewrite rules, whereas the event structures are a true concurrency model. Irrespective of this aspect we give a result for the encoding that shows that the concurrency embodied by the event structure is captured in the translation psi-process through the standard interleaving diamond. Another feature of true concurrency models is that they are well behaved wrt. action refinement. For this we give another result showing that action refinement is preserved by our translation; under a properly defined refining function on psi-process, which we define similarly to the refinement function on the event structures. A corollary of the refinement is that we get a composition result for a restricted form of parallel operation on event structures.

We believe that the techniques that we use in the translation of event structures can be easily extended to translate condition-response event structures (CRES) and DCR-graphs into instances of psi-calculi.

1 Psi-calculi and event structures

We recall only the notions from psi-calculi and event structures that we use in this short version. A long version containing complete proofs, more definitions and explanations can be found on the authors homepage (`http://folk.uio.no/haakno/`).

A Psi-calculus is built over *data* terms M, N. These are used in the communication primitives as $\overline{M}N.P$ to say that the process sends data N over the channel M, and $\underline{K}(\lambda\widetilde{x})L.Q$, to say that channel K receives data matching the pattern $\lambda\widetilde{x}.L$. Interaction is under the conditions:
(1) The two channels M and K are equivalent, as defined by a predicate $M \leftrightarrow K$
(2) N matches the input pattern, i.e., $N = L[\widetilde{x} := \widetilde{T}]$ for some sequence of terms $\widetilde{T}$

Psi-calculi are parametrised by the following entities, and we instantiate these to obtain our particular psi-calculus for capturing event structures in Section 2. Define three nominal data types: $\mathbf{T}$, i.e., the (data) terms ranged over by N, M; $\mathbf{C}$, i.e., the conditions ranged over by φ; $\mathbf{A}$, i.e., the assertions ranged over by Ψ. Define four equivariant operators: $\dot{\leftrightarrow}\colon \mathbf{T} \times \mathbf{T} \to \mathbf{C}$, *channel equivalence*; $\otimes : \mathbf{A} \times \mathbf{A} \to \mathbf{A}$, *composition*; $\mathbf{1} : \mathbf{A}$, *unit*; $\vdash \subseteq \mathbf{A} \times \mathbf{C}$, *entailment*; respecting the following conditions: $\dot{\leftrightarrow}$ must be symmetric and transitive; $\otimes$ must be associative, commutative, compositional, and maintain identity.

*I would like to thank my supervisors Thomas Hildebrandt and Cristian Prisacariu for their help.

Transitions are of kind $\Psi \triangleright P \xrightarrow{\alpha} P'$, meaning that when the environment provides the assertion Ψ then a well formed agent P can do an α to become P'. The part of the operational semantics used in this paper is given below, where Ψ_Q in PAR is the assertion exposed by Q.

$$\frac{\Psi \vdash M \leftrightarrow K}{\Psi \triangleright \overline{M}N.P \xrightarrow{\overline{K}N} P}\ \textsc{Out} \qquad \frac{\Psi \triangleright P_i \xrightarrow{\alpha} P' \qquad \Psi \vdash \varphi_i}{\Psi \triangleright case\ \widetilde{\varphi} : \widetilde{P} \xrightarrow{\alpha} P'}\ \textsc{Case} \qquad \frac{\Psi_Q \otimes \Psi \triangleright P \xrightarrow{a} P'}{\Psi \triangleright P|Q \xrightarrow{a} P'|Q}\ \textsc{Par}$$

For event structures we follow that notation of [NPW81].

Definition 1.1 (prime event structures)**.** *A prime event structure is a tuple $\varepsilon = (E, <, \sharp)$ where E is a set of events, $< \subseteq E \times E$ is an (irreflexive) partial order (the causality relation) satisfying the principle of finite causes, i.e., $\forall e \in E : \{d \in E | d < e\}$ is finite, $\sharp \subseteq E \times E$ is an irreflexive, symmetric relation (the conflict relation) satisfying the principle of conflict heredity, i.e., $\forall d, e, f \in E : d < e \wedge d\sharp f, \Rightarrow e\sharp f,$.*

A prime event structure models a concurrent system, intuitively, by using $d < e$ to mean that d is a prerequisite of e, and $d\sharp e$ to mean that d and e cannot both happen in same run, i.e., a choice/branching point. Casual independence (concurrency) between events $d||e$ is modelled as the absence of casual dependence or conflict, i.e., $\neg(d < e \vee e < d \vee d\sharp e)$. The computation of an event structure ε is captured by the subsets of events, called configurations C_ε, each containing the events that happened in some partial run.

2 Encoding event structures in psi-calculi

Definition 2.1 (event psi-calculus)**.** *We define a psi-calculus instance, which we call* event psi*, parametrized by a nominal set E, to be understood as* events*. This means providing the following definitions of the key elements of a psi-calculus instance:*

$$\boldsymbol{T} \stackrel{def}{=} E \qquad \boldsymbol{C} \stackrel{def}{=} \mathcal{P}(E) \times \mathcal{P}(E) \qquad \boldsymbol{A} \stackrel{def}{=} \mathcal{P}(E) \qquad \dot{\leftrightarrow} \stackrel{def}{=} = \qquad \otimes \stackrel{def}{=} \cup \qquad \boldsymbol{1} \stackrel{def}{=} \emptyset$$

$$\vdash \stackrel{def}{=} \Psi \vdash \varphi \text{ iff } (\pi_L(\varphi) \subseteq \Psi) \wedge (\Psi \cap \pi_R(\varphi) = \emptyset) \qquad \Psi \vdash a \dot{\leftrightarrow} b \text{ iff } a = b$$

We have that T, C, and A are nominal sets. Channel equivalence maintains symmetry and transitivity since $=$ is upholding these rules. The $\otimes$ is compositional, associative and commutative as $\cup$ is. and as $\emptyset \cup S = S$, for any set S. We have that identity is maintained.

We use only one simple nominal set for T which intuitively is understood to be the set of all events. The conditions C consists of pairs of subsets of events. The assertions A is intuitively understood as capturing the set of all executed events (and thus ranges over T). Channel equivalence is equality of event names. Composition of two assertions is the union of the sets. Identity is the empty-set. The entailment $\vdash$ intuitively captures when events may fire, thus describing when events are enabled by a configuration in event structures, as well as how it affects channel equivalence.

Definition 2.2 (from event structures to event psi)**.** *We define a function* PSI *which given an event structure $\varepsilon = (E, <, \sharp)$ and a Configuration C_ε of ε, returns an event psi process $P_E = |_{e \in E} P_e$ with $P_e = (\!|\{e\}|\!)$ if $e \in C_\varepsilon$, otherwise $P_e = case\ \varphi_e : \overline{e}e.(\!|\{e\}|\!)$, where we have $\varphi_e = (<e, \sharp e)$ with $<e = \{e' | (e', e) \in <\}$ the set of all events e has as conditions, and $\sharp e = \{e' | (e', e) \in \sharp\}$ the events e is in conflict with.*

A process generated by the PSI function is built up by smaller event processes put in parallel. Where each event process is either in state executed or not executed depending on whether it is in the configuration or not.

For each event e we make a condition φ_e that contains two sets, the set of events e is depending on and the set of events e is in conflict with. When an event happens we will have a transition over the channel with the same name as the event. For event structures where the configuration is empty we only give the tuple as input.

Lemma 2.1 (correspondence configuration–frame)**.** *For any event structure ε and configuration C_ε the frame of the event-psi process $\text{PSI}(\varepsilon, C_\varepsilon)$ corresponds to the configuration C_ε.*

Lemma 2.2 (transitions maintain configurations)**.** *For some event structure ε and some configuration of it C_ε, then any transition from this configuration $C_\varepsilon \xrightarrow{e} C'_\varepsilon$ is matched by a transition $\text{PSI}(\varepsilon, C_\varepsilon) \xrightarrow{\overline{e}e} \text{PSI}(\varepsilon, C'_\varepsilon)$ in the corresponding psi process.*

Theorem 2.3 (preserving concurrency)**.** *For an event structure $(E, <, \sharp)$ with two concurrent events $e||e'$ then in the translation $\text{PSI}(E, <, \sharp)$ we find the behaviour forming the interleaving diamond, i.e., $\emptyset \rhd \text{PSI}(E, <, \sharp) \xrightarrow{e} P_1 \xrightarrow{e'} P_2$ and $\emptyset \rhd \text{PSI}(E, <, \sharp) \xrightarrow{e'} P_3 \xrightarrow{e} P_4$ with $P_2 = P_4$.*

We want to be able to refine the psi processes on the same line as event structures are refined in [vGG01]; i.e., for labeled prime event structures, a function $ref : Act \to \mathbf{E}_{prime}$ is called a refinement function iff $\forall a \in Act : ref(a)$ is non-empty, finite and conflict-free. Then ref(ε) is the prime event structure defined by: $E_{ref(\varepsilon)} := \{(e, e')|e \in E_\varepsilon, e' \in E_{ref(l_\varepsilon(e))}\}$; $(d, d') <_{ref(\varepsilon)} (e, e')$ iff $d <_\varepsilon e$ or $(d = e \wedge d' <_{ref(l_\varepsilon(d))} e')$; $(d, d')\sharp_{ref(\varepsilon)}(e, e')$ iff $d\sharp_\varepsilon e$; $l_{ref(\varepsilon)}(e, e') := l_{ref(l_\varepsilon(e))}(e')$.

Definition 2.3. *We define a function ref^P that refines an event-psi process to a new one over $T^P = \{(e, e')|e \in T, e' \in T_{ref(e)}\}$, with $\varphi^P = \{(<(e, e'), \sharp(e, e'))|(e, e') \in T^P\}$, where $<(e, e') = \{(d, d')|d \in < e \vee d = e \wedge d' \in <_{ref(d)} e\}$ and $\sharp(e, e') = \{(d, d')|d \in \sharp e\}$ to obtain $P_{ref} = |_{(e,e')\in T^P} P_{(e,e')}$ with $P_{(e,e')} = (\!|\{(e, e')\}|\!)$ if $e \in \Psi_P$, else $P_{(e,e')} = \overline{(e, e')}(e, e').(\!|\{(e, e')\}|\!)$.*

The new names are all possible pairs of a parent events name and one of its children events name. This can be the same as the parents name. We make new conditions for each of the new names (e, e'), where $<(e, e')$ is all pairs of names where first part is a condition for e, or if first part is same as e, second part is condition for e'. For $\sharp(e, e')$ we have that it is all pairs of names where first part is precondition for e. Then make new event processes for each new pair, where state is either executed or not executed depending on whether first part of event name was in the frame of the old event-psi.

Theorem 2.4 (refinement of event-psi corresponds to refinement in ES)**.** *For a prime event structure ε we have that:* $\text{PSI}(ref(\varepsilon)) = ref^P(\text{PSI}(\varepsilon))$.

References

[BJPV11] Jesper Bengtson, Magnus Johansson, Joachim Parrow, and Björn Victor. Psi-calculi: a framework for mobile processes with nominal data and logic. *Logical Methods in Computer Science*, 7(1), 2011.

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

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

ST-Configuration Structures

Cristian Prisacariu

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

1 Introduction

The present paper defines ST-structures. The main purpose is to provide concrete relationships between highly expressive concurrency models coming from two different schools of thought: the higher dimensional automata (HDA), a *state-based* approach of Pratt and van Glabbeek; and the configuration structures and (in)pure event structures, an *event-based* approach of van Glabbeek and Plotkin. In this respect we make comparative studies of the expressive power of ST-structures relative to the above models. Moreover, standard notions from other concurrency models can be defined for ST-structures, like steps and paths, bisimulation, and action refinement, and related results can be found in the extended version. These investigations of ST-structures are intended to provide a better understanding of the *state-event duality* described by Pratt, and also of the (a)cyclic structures of higher dimensional automata.

ST-configuration structures are a natural extension of configuration structures to the setting of higher dimensional automata. Configuration structures [6] are used in [5] as the most expressive model of concurrency which has a natural way of defining refinement and the partial order bisimulations, like history preserving bisimulation. The notion of an ST-configuration has been used in [7] to define ST-bisimulation and in [4] in the context of *HDA*. But the model of *ST-configuration structures*, as we define here for capturing concurrency, does not appear elsewhere. ST-structures have the power to look at the currently executing concurrent events. This captures a main characteristic of higher dimensional automata, which cannot be captured by standard models like configuration structures.

2 ST-configuration structures

Definition 2.1 (ST-configuration)**.** *An ST-configuration is a pair of sets (S, T) with the following property:*

$$\textit{(start before terminate)}\quad T \subseteq S.$$

Intuitively S contains the events that have started and T the event that have terminated.

Definition 2.2 (ST-configuration structures)**.** *An* ST-configuration structure *(also called* ST-structure*) is a tuple $\mathsf{ST} = (ST, l)$ with ST a set of ST-configurations and l a labelling function of the events, $l : \bigcup_{S \in ST} S \to \Sigma$, with ST satisfying the constraint: if $(S, T) \in ST$ then $(S, S) \in ST$.*

The constraint above is a closure so that we do not represent events that are started but never terminated.

ST-structures have a natural *computational interpretation* (on the same lines as configuration structures) as *steps* between ST-configurations, and *paths*. Our results show that this computational interpretation is more fine-grained than for other models we compare with. Intuitively, opposed to standard event-based models, the computational interpretation of ST-structures naturally captures the "during" aspect of the events, i.e., what happens while an

event is executing (before it has finished). Action refinement and bisimulation are well behaved wrt. this interpretation. The model of *HDAs* do the same job but in the state-based setting. Besides, ST-structures exhibit a natural *observable information* (on the same lines as for *HDAs*) as *ST-traces*, which, cf. [4, Sec.7.3], constitute the best formalization of observable content.

Definition 2.3 (ST steps)**.** *A step between two ST-configurations is defined as either:*
s-step $(S,T) \xrightarrow[s]{a} (S',T')$ *when* $T = T'$, $S \subset S'$, $S' \setminus S = \{e\}$ *and* $l(e) = a$; *or*
t-step $(S,T) \xrightarrow[t]{a} (S',T')$ *when* $S = S'$, $T \subset T'$, $T' \setminus T = \{e\}$ *and* $l(e) = a$.
When the type is unimportant we denote a step by $\xrightarrow{a}$ *for* $\xrightarrow[s]{a} \cup \xrightarrow[t]{a}$.

Definition 2.4 (stable ST-structures)**.** *An ST-structure* $\mathsf{ST} = (ST, l)$ *is called:*

1. rooted *iff* $(\emptyset,\emptyset) \in ST$;
2. connected *iff for any non-empty* $(S,T) \in ST$, *either* $\exists e \in S : (S \setminus e, T) \in ST$ *or* $\exists e \in T : (S, T \setminus e) \in ST$;
3. closed under bounded unions *(respectively* bounded intersections*) iff for any* (S,T), (S',T'), $(S'',T'') \in ST$ *s.t.* $(S,T) \cup (S',T') \subseteq (S'',T'')$, *then* $(S,T) \cup (S',T') \in ST$ *(rsp.* $(S,T) \cap (S',T') \in ST$*).*

An ST-structure is called stable *iff it is rooted, connected, and closed under bounded unions and intersections.*

Definition 2.5 (adjacent-closure)**.** *An ST-structure* ST *is* adjacent-closed *if the following are respected:*

1. *if* $(S,T),(S \cup e,T),(S \cup \{e,e'\},T) \in \mathsf{ST}$, *with* $(e \neq e') \notin S$, *then* $(S \cup e',T) \in \mathsf{ST}$;
2. *if* $(S,T),(S \cup e,T),(S \cup e,T \cup e') \in \mathsf{ST}$, *with* $e \notin S \wedge e' \notin T \wedge e \neq e'$, *then* $(S,T \cup e') \in \mathsf{ST}$;
3. *if* $(S,T),(S \cup e,T),(S,T \cup e') \in \mathsf{ST} : e \notin S \wedge e' \notin T \wedge e \neq e'$, *then* $(S \cup e,T \cup e') \in \mathsf{ST}$;
4. *if* $(S,T),(S,T \cup e),(S,T \cup \{e,e'\}) \in \mathsf{ST}$, *with* $(e \neq e') \notin T$, *then* $(S,T \cup e') \in \mathsf{ST}$.

Knowing the definition of higher dimensional automata (see [1,2,4]) one can see a correlation of the above definition of adjacent-closure on ST-structures and the cubical laws of higher dimensional automata. This correlation is even more visible in the definition of *adjacency* of [4, Def.19] which is used to define homotopy over higher dimensional automata. Since homotopy classes essentially define histories, then the above adjacent-closure on ST-structures intuitively makes sure that the histories of ST-configurations are not missing anything.

The standard example of the square with the empty inside is adjacent-closed but not closed under unions nor under intersections. The example of the parallel switch of Winskel [8] is adjacent-closed and closed under unions, but not closed under intersections (can be pictured as only three sides of a cube in *HDA*).

Definition 2.6 (concurrency and causality)**.** *For a particular ST-configuration* $(S,T) \in \mathsf{ST}$ *define the relations of* concurrency *and* causality *on the events in* S *as:*

concurrency *for* $e,e' \in S$ *then* $e||e'$ *iff exists* $(S',T') \subseteq (S,T)$ *s.t.* $(S',T') \in \mathsf{ST}$ *and* $\{e,e'\} \subseteq S' \setminus T'$;

causality *for* $e,e' \in S$ *then* $e < e'$ *iff* $e \neq e' \wedge \forall (S',T') \subseteq (S,T) : (S',T') \in \mathsf{ST} \Rightarrow (e' \in S' \Rightarrow e \in T')$.

ST-structures represent *concurrency* in a way that is different than other event-based models in the sense that each ST-configuration gives information about the currently concurrent events, and this information is persistent throughout the whole execution. Two events are considered concurrent wrt. a particular ST-configuration if and only if at some point in the past (i.e., in some sub-configuration) both events appeared as executing (i.e., in S') and none was terminated yet (i.e., not in T'); they were both executing concurrently. In event structures or configuration structures in order to decide whether two events are concurrent one needs to look at many configurations or many events to decide this. For example, in event structures the concurrency is defined as not being dependent nor conflicting; which requires to inspect all configurations to decide. An ST-configuration does not give complete information about the concurrency relation in the whole system. In consequence one could view the information about concurrency that an ST-configuration provides as being sound but not complete.

On arbitrary ST-structures the concurrency and causality are not interdefinable (in a standard way e.g. [5, Def.5.6] where concurrency is the negation of causality). Nevertheless, concurrency and causality are disjoint on every ST-configuration of an arbitrary ST-structure. For the more well behaved stable ST-structures the concurrency and causality are interdefinable. Even more, results similar to the ones in [5, Sec.5.3] can be stated and proven about stable ST-structures and their causality partial order.

Definition 2.7 (cf. [5, Def.5.1] [6, Def.1.1])**.** *A configuration structure $\mathsf{C} = (E, C)$, is formed of a set E of events and a set of configurations which are subsets of events $C \subseteq 2^E$. A* labeled *configuration structure also has a labeling function of its events, $l : E \to \Sigma$.*

Proposition 2.8. *If an ST-structure ST is rooted, connected, or closed under bounded unions, or intersections, then the corresponding $\mathsf{C}(\mathsf{ST})$ is respectively rooted, connected, closed under bounded unions, or intersections. The mapping $\mathsf{C} : \mathbb{ST} \to \mathbb{C}$ is defined to associate to every ST-structure ST a configuration structure by keeping only those ST-configurations that have $S = T$; i.e., $\mathsf{C}(\mathsf{ST}) = \{T \mid (S, T) \in \mathsf{ST} \wedge S = T\}$, which preserves the labeling.*

Theorem 2.9. *Configuration structures are strictly embedded into ST-structures. Define a mapping $\mathsf{ST} : \mathbb{C} \to \mathbb{ST}$ that associates to every configuration structure C an ST-structure $\mathsf{ST}(\mathsf{C})$ by associating to each configuration $X \in \mathsf{C}$ an ST-configuration $\mathsf{ST}(X) = (X, X) \in \mathsf{ST}(\mathsf{C})$ and for each transition $X \to_C Y \in \mathsf{C}$ an ST-configuration $\mathsf{ST}(X \to_C Y) = (Y, X) \in \mathsf{ST}(\mathsf{C})$. This map ST preserves the asynchronous concurrent steps of the configuration structure, i.e., for each asynchronous step $X \to_C Y \in \mathsf{C}$ there is a chain of single steps in the ST-structure $\mathsf{ST}(\mathsf{C})$ that passes through (Y, X) (thus signifying the concurrent execution of all events in $Y \setminus X$).*

Corollary 2.10. *An ST-structure $\mathsf{ST}(\mathsf{C})$ generated as in Theorem 2.9 is adjacent-closed (though not necessarily closed under bounded unions nor intersections).*

Corollary 2.11. *In an ST-structure $\mathsf{ST}(\mathsf{C})$ generated as in Theorem 2.9 the ST-configurations with $S = T$ correspond exactly to the configurations of C. That is to say that $\mathsf{C}(\mathsf{ST}(\mathsf{C})) \cong \mathsf{C}$.*

Corollary 2.12. *The ST-structure obtained in Th. 2.9 is* "filled in"*, in the sense that any cube is filled in. By a "cube" it is meant an initial ST-configuration (S, S), a final $(S \cup X, S \cup X)$, where X is a nonempty set of events, together with all the ST-configurations (Y, Y) from the subsets $S \subseteq Y \subseteq S \cup X$. To be "filled in" means that the intermediate ST-configuration $(S \cup X, S)$ exists.*

But there is not a one to one correspondence between ST-structures and the configuration structures because there can be several ST-structures that have the same configuration structure. The example is of one *HDA* square that is filled in and one that is not; both have the

same set of corners and hence the same configuration structure. But the two ST-structures are not isomorphic and also not hh-bisimilar.

Proposition 2.13. *For stable and adjacent-closed ST-structures and stable configuration structures there is a one-to-one correspondence. (The adjacency is necessary.)*

Similar connections are investigated wrt. the *(impure) event structures* of [6] and wrt. the higher dimensional automata of [3, 4].

References

[1] V. R. Pratt. Modeling concurrency with geometry. In *POPL'91*, pages 311–322, 1991.

[2] V. R. Pratt. Higher dimensional automata revisited. *MSCS*, 10(4):525–548, 2000.

[3] V. R. Pratt. Transition and Cancellation in Concurrency and Branching Time. *MSCS*, 13(4):485–529, 2003.

[4] R. van Glabbeek. On the Expressiveness of Higher Dimensional Automata. *TCS*, 356(3):265–290, 2006.

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

[6] R. van Glabbeek & G. Plotkin. Configuration structures, event structures and Petri nets. *TCS*, 410(41):4111–4159, 2009.

[7] R. van Glabbeek & F. Vaandrager. The Difference between Splitting in n and n+1. *Inf. Comput.*, 136(2):109–142, 1997.

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

Lock-Polymorphic Behaviour Inference for Deadlock Checking

Ka I Pun, Martin Steffen and Volker Stolz

Department of Informatics, University of Oslo, Norway

Deadlocks are a common problem for concurrent programs with shared resources. According to the classic characterization from [2], a deadlocked state is marked by a number of processes forming a cycle where each process, unwilling to release its own resource, is waiting on the resource held by its neighbor. The inherent non-determinism make deadlocks, as other errors in the presence of concurrency, hard to detect and to reproduce. We present a static analysis using behavioral effects to detect deadlocks in a higher-order concurrent calculus. Deadlock freedom, an important safety property for concurrent programs, is a thread-global property, i.e., the blame for a deadlock in a defective program cannot be put on a single thread, it is two or more processes that share responsibility; the somewhat atypical situation, where a process forms a deadlock with itself, cannot occur in our setting, as we assume re-entrant locks. The approach presented in this paper works in two stages: in a first stage, an effect-type system uses a static behavioral abstraction of the codes' behavior, concentrating on the lock interactions. To analyze the consequences on the global level, in particular for detecting deadlocks, the combined individual abstract thread behaviors are explored in the second stage.

Two challenges need to be tackled to make such a framework applicable in practice. For the first stage on the thread local level, the static analysis must be able to *derive* the abstract behavior, not just check compliance of the code with a user-provided description. This is the problem of type and effect *inference* or reconstruction. As usual, the abstract behavior needs to over-approximate the concrete one which means, concrete and abstract description are connected by some *simulation* relation: everything the concrete system does, the abstract one can do as well (modulo some abstraction function relating the concrete and abstract states). For the second stage, exploring the (abstract) state space on the global level, obtaining *finite* abstractions is crucial. In our setting, there are four principal sources of infinity: the calculus, 1) allowing recursion, supports 2) dynamic thread creation, 3) dynamic lock creation, and 4) with re-entrant locks, the lock counters are unbounded. Our approach offers sound abstractions for the mentioned sources of unboundedness, except that we do not have an abstraction usable for deadlock detection in the presence of dynamic thread creation. We shortly present in a non-technical manner the ideas behind the abstraction.

Effect inference on the thread local level

As mentioned, in the first stage of the analysis, the analysis uses a behavioral type and effect system to over-approximate the lock-interactions of a single thread. To force the user to annotate the program with the expected behavior in the form of effects is impractical, so the type and especially the behavior should be inferred automatically. Effect inference, including inferring behavioral effects, has been studied earlier and applied to various settings, including obtaining static over-approximations of behavior for concurrent languages by Amtoft, Nielson and Nielson [1]. We apply effect inference to deadlock detection and as is standard (cf. e.g. [8, 11, 1]), the inference system is constraint-based, where the constraints in particular express an approximate order between behaviors. Besides being able to infer the behavior, it is important that the static approximation is as precise as possible. Since our calculus supports

higher-order functions, it is thus important that the analysis may distinguish different instances of a function body depending on their calling context, i.e., the analysis should be *polymorphic* or *context-sensitive.* This can be seen as an extension of let-polymorphism to effects and using constraints. The effect reconstruction resembles the known type-inference algorithm for let-polymorphism by Damas and Milner [4, 3] and this has been used for effect-inference in various settings, e.g., in the works mentioned above.

Deadlock checking in our earlier work [9] was not polymorphic (and we did not address effect inference). The extension in this paper leads to an increase in precision wrt. checking for deadlocks, as illustrated by the small example below, where the two lock creation statements are labeled by π_1 and π_2:

```
let l$_1$ = new$^{\flab_1}$ L in let l$_2$ = new$^{\flab_2}$ L in
let f = fn x:L . ( x.lock; x.lock )
in spawn(f(l$_1$)); f(l$_2$)
```

The main thread, after creating two locks and defining function f, spawns a thread, and afterward, the main thread and the child thread run in parallel, each one executing an instance of f with different actual lock parameters. In a setting with re-entrant locks, the program is obviously deadlock-free. Part of the type system of [9] determines the potential origin of locks by data-flow analysis. When analyzing the body of the function definition, the analysis cannot distinguish the two instances of f (the analysis is context-*insensitive*). This inability to distinguish the two call sites —the "context"— forces that the type of the formal parameter is, at best, $\mathtt{L}^{\{\pi_1,\pi_2\}}$, which means that the lock-argument of the function is potentially created at either point. Based on that approximate information, a deadlock looks possible through a "deadly embrace" [5] where one thread takes first lock π_1 and then π_2, and the other thread takes them in reverse order , i.e., the analysis would report a (spurious) deadlock. The context-sensitive analysis presented here correctly analyzes the example as deadlock-free.

Deadlock preserving abstractions on the global level

Lock abstraction For dynamic data allocation, a standard abstraction is to *summarize* all data allocated at a given program point into one abstract representation. In the presence of loops or recursion, the abstracting function mapping concrete locks to their abstract representation necessarily is non-injective. For concrete, ordinary programs it is clear that identifying locks may change the behavior of the program. What makes identification of locks in general tricky, and here in particular connection with deadlocks, is that, on the one hand. it leads to *less* steps, in that lock-protected critical sections may become larger, and on the other hand to *more* steps at the same time, in that deadlocks may disappear when identifying locks. That this form of summarizing lock abstraction is problematic when analyzing properties of concurrent programs has been observed elsewhere as well, cf. e.g. Kidd et al. in [7].

For a sound abstraction for deadlock detection when identifying locks in the described way, one faces thus the following dilemma: a) the abstract level, using the abstract locks, need to show at least the behavior of the concrete level, i.e., we expect they are related by a form of simulation. On the other hand, to preserve not only the possibility of doing steps, but also *deadlocks,* the opposite must hold sometimes: a) a concrete program waiting on a lock and unable to make a step thereby, must imply an analogous situation on the abstract level, lest we should miss deadlocks. Let's write $l, l_1, l_2, \ldots$ for concrete lock references and $\pi, \pi', \ldots$ for program points of lock creation, i.e., abstract locks. To satisfy a): when a concrete program takes a lock, the abstract one must be able to "take" the corresponding abstract lock, say π. A consequence of a) is that taking an abstract lock is always enabled. That is consistent with the

abstraction as described where the abstract lock π confuses an arbitrary number of concrete locks including e.g., those freshly created, which may be taken.

Consequently, abstract locks loose their "mutual exclusion" capacity: where a concrete heap is a mapping which associates to each lock references the number of times *at most* one process is holding it, an abstract heap $\hat{\sigma}$ then records how many times an abstract lock π is held by the various processes, e.g. three times by one process and two times by another. The corresponding natural number of the abstractly represent the *sum* of the lock values of all concrete locks (per process). Without ever blocking, the abstraction leads to more possible steps, but to cater for b), the abstraction still needs to appropriately define, given an abstract heap and an abstract lock π, when a process waits on the abstract lock, as this may indicate a deadlock. The definition basically has to capture all possibilities of waiting on one of the corresponding concrete locks. The sketched intuitions to obtain a sound abstract summary representation for locks and correspondingly for heaps lead also to a corresponding refinement of "over-approximation" in terms of simulation: not only must the a) positive behavior be preserved as in standard simulation, also the possibility of waiting on a lock and ultimately possibility of deadlock needs to be preserved. For this we introduce the notion of *deadlock sensitive* simulation. The definition is analogous to the one from [9]. However, it takes into account now that the analysis is polymorphic and the definition is no longer based on an direct operational interpretation of the behavior of the effects. Instead it is based on the behavioral constraints used in the inference systems.

Counter abstraction and further behavior abstraction Two remaining causes of an infinite state space are the values of lock counters, which may grow unboundedly and the fact that, for each thread, the effect behavior represent abstractly the *stack* of function calls for that thread. With sequential composition as construct for abstract behavioral effects allows to represent non-tail-recursive behavior, corresponding to the context-free call-and-return behavior of the underlying program. To curb that source of infinity, we allow to replace the behavior by a tail-recursive over-approximation. The precision of the approximation can be adapted in choosing the depth of calls after which the call-structure collapses into arbitrary, chaotic behavior. A finite abstraction for the lock-counters is achieved similarly by imposing an upper bound on the considered lock counter, beyond which the locks behave non-deterministically. Again, for both abstractions it is crucial, that the abstraction preserves also deadlocks, which we capture again using the notion of deadlock-sensitive simulation.

Compared to [9], the paper makes the following contributions: 1) the effect analysis is generalized to a context-sensitive formulation, using constraint, for which we provide 2) an inference algorithm. Finally, 3) we allow summarizing multiple concrete locks into abstract ones, while still preserving deadlocks. All technical materials, lemmas and proofs can be found in the technical report [10].

References

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

[2] E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2), 1971.

[3] L. Damas. *Type Assignment in Programming Languages.* PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1985. CST-33-85.

[4] L. Damas and R. Milner. Principal type-schemes for functional programming languages. In *Ninth Annual Symposium on Principles of Programming Languages (POPL) (Albuquerque, NM)*, pages 207–212. ACM, January 1982.

[5] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [6].

[6] F. Genyus. *Programming Languages.* Academic Press, 1968.

[7] N. Kidd, T. W. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, 2011.

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

[9] K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. *Journal of Logic and Algebraic Programming*, 81(3):331–354, 2012. A preliminary version was published as University of Oslo, Dept. of Computer Science Technical Report 404, March 2011.

[10] K. I. Pun, M. Steffen, and V. Stolz. Lock-polymorphic behaviour inference for deadlock checking. Technical report 436, University of Oslo, Dept. of Informatics, Sept. 2013. available electronically at `http://www.ifi.uio.no/~msteffen/download/13/lockpolymorphic-rep.pdf`.

[11] J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.

Structural Congruences for Bialgebraic Semantics

Jurriaan Rot and Marcello Bonsangue

LIACS — Leiden University
{j.c.rot, m.m.bonsangue}@liacs.leidenuniv.nl

Summary. It was observed by Turi and Plotkin that structural operational semantics can be studied at the level of universal coalgebra, providing specification formats for well-behaved operations on many different types of systems. We extend this framework with non-structural assignment rules which can express, for example, the syntactic format for structural congruences proposed by Mousavi and Reniers. Our main result is that the operational model of such an extended specification is well-behaved, in the sense that bisimilarity is a congruence and that bisimulation-up-to techniques are sound.

Background. Structural operational semantics (SOS) is a framework for defining the semantics of programming languages and calculi in terms of transition system specifications. By imposing syntactic restrictions, one can prove well-behavedness properties of transition systems at the meta-level of their specification. For instance, any specification in the GSOS format [1] has a unique operational model, on which bisimilarity is a congruence.

Traditionally, research in SOS has focused on labelled transition systems as the fundamental model of behaviour. Turi and Plotkin [14] introduced the *bialgebraic* approach to structural operational semantics, where in particular GSOS can be studied at the level of *universal coalgebra* [11]. The theory of coalgebras provides a mathematical framework for the uniform study of many types of state-based systems, including labelled transition systems but also, e.g., (non)-deterministic automata, stream systems and various types of probabilistic and weighted automata. In the coalgebraic framework, there is a canonical notion of bisimilarity, which instantiates to the classical definition of (strong) bisimilarity in the case of labelled transition systems. It is shown in [14] that GSOS specifications can be generalised by certain natural transformations, which are called *abstract GSOS specifications*, and that these correspond to the categorical notion of *distributive laws*. This provides enough structure to prove at this general level that bisimilarity is a congruence. By instantiating the theory to concrete instances, one can then obtain congruence formats for systems such as probabilistic automata, weighted transition systems and streams — see [5] for an overview. Another advantage of abstract GSOS is that bisimulation up to context is "compatible" [10, 9], providing a sound enhancement of the bisimulation proof method which can be combined with other compatible enhancements such as bisimulation up to bisimilarity [12, 8].

Adding assignment rules. In this paper we consider non-structural rules such as the following:

$$\frac{!x \mid x \xrightarrow{a} t}{!x \xrightarrow{a} t} \tag{1}$$

The rule in (1) properly defines the replication operator in CCS[1]: intuitively $!x$ represents $x \mid x \mid x \mid \ldots$, i.e., the infinite parallel composition of x with itself. In fact, the above rule

[1] The simpler rule $\frac{x \to x'}{!x \to !x \mid x'}$ is problematic in the presence of the sum operator [8, 13].

can be seen as assigning the behaviour of the term $!x \mid x$ to the simpler term $!x$, therefore we call it an *assignment rule*. Being inherently non-structural, such an assignment rule cannot directly be embedded in the bialgebraic framework of Turi and Plotkin, where the behaviour of terms is computed inductively. In this paper we show how to interpret assignment rules together with abstract GSOS specifications. As it turns out, this requires the assumption that the functor which represents the type of coalgebra is *ordered* as a complete lattice; for example, in the case of labelled transition systems this order is simply inclusion of sets of pairs (a, x) of a label a and a state x. The operational model on closed terms then is the *least* model such that every transition either (1) can be derived from a rule in the specification, or (2) there is a rule assigning to an operator σ the behaviour of a term t in the model. To ensure the existence of such least models, we restrict to *monotone* abstract GSOS specifications, a generalisation of the *positive GSOS format* for transition systems [3]. Positive GSOS can be seen as the greatest common divisor of GSOS and the tyft/tyxt format.

Our main result is that the interpretation of a monotone abstract GSOS specification together with a set of assignment rules is itself the operational model of another (typically larger) abstract GSOS specification. Like the interpretation of a GSOS specification with assignment rules, we construct this latter specification by fixpoint induction. As a direct consequence of this alternative representation of the interpretation, we obtain that bisimilarity is a congruence and that bisimulation up to context is sound and even compatible — properties that do not follow from bisimilarity being a congruence [8]. As an example application, we obtain the compatibility of bisimulation-up-to techniques for CCS with replication, which so far had to be shown with an ad-hoc argument [8].

Structural congruences. A further contribution of this work consists in combining *structural congruences* [6, 7] with the bialgebraic framework using assignment rules. Structural congruences were introduced in the operational semantics of the π-calculus in [6]. The basic idea is that SOS specifications are extended with *equations* on terms, which are then linked by a special deduction rule. This rule essentially states that if two processes are equated by the congruence generated by the set of equations, then they can perform the same transitions. Prototypical examples are the specification of the parallel operator by combining a single rule with commutativity, and the specification of the replication operator by an equation, both shown below:

$$\frac{x \xrightarrow{a} x'}{x \mid y \xrightarrow{a} x' \mid y} \quad x \mid y = y \mid x \qquad\qquad !x = !x \mid x \tag{2}$$

In [7] Mousavi and Reniers show that SOS rules with structural congruences can be interpreted in different but equivalent ways. They exhibit very simple examples of equations and SOS rules for which bisimilarity is not a congruence, even when the SOS rules are in the tyft (or the GSOS) format. As a solution to this problem they introduce a restricted format for equations, called cfsc, for which bisimilarity is a congruence when combined with tyft specifications.

In the present work we show how to interpret structural congruences at the general level of coalgebras, in terms of an operational model on closed terms. We prove that when the equations are in the cfsc format then they can be encoded by assignment rules, in such a way that their respective interpretations coincide up to bisimilarity. Consequently, not only is bisimilarity a congruence for monotone abstract GSOS combined with cfsc equations, but also bisimulation up to context and bisimilarity is compatible.

Related work. The main work on structural congruences [7] focuses on labelled transition systems, whereas our work considers the more general notion of coalgebras. As for transition

systems, the basic rule format in [7] is tyft/tyxt[2], which is strictly more general than positive GSOS since it allows lookahead. However, while [7] proves congruence of bisimilarity this does not imply the compatibility (or even soundness) of bisimulation up to context [8], which we obtain in the present work (and is in fact problematic in the presence of lookahead).

In the bialgebraic setting, Klin [4] showed that by moving to CPPO-enriched categories, one can interpret recursive constructs which have a similar form as our assignment rules. Technically our approach is different, allowing us to stay in the familiar category of sets, and apply the coalgebraic bisimulation-up-to techniques which are based in this category. Further, in [4] each operator is either specified by an equation or by operational rules, disallowing a specification such as that of the parallel operator in (2).

In [2] it is shown how to obtain a distributive law for a monad which is obtained as the quotient of another monad by imposing equations on terms, under the condition that the distributive law respects the equations. However, this condition requires that the equations already hold semantically, which is fundamentally different from the present work where we define behaviour by imposing equations on an operational specification.

Acknowledgements. We would like to thank Daniel Gebler, Bartek Klin and Jan Rutten for helpful suggestions and discussions. Our research has been funded by the Netherlands Organisation for Scientific Research (NWO), CoRE project, dossier number: 612.063.920.

References

[1] B. Bloom, S. Istrail, and A. Meyer. Bisimulation can't be traced. *J. ACM*, 42(1):232–268, 1995.

[2] M. M. Bonsangue, H. H. Hansen, A. Kurz, and J. Rot. Presenting distributive laws. In R. Heckel and S. Milius, editors, *CALCO*, volume 8089 of *LNCS*, pages 95–109. Springer, 2013.

[3] M. Fiore and S. Staton. Positive structural operational semantics and monotone distributive laws. In *CMCS Short Contributions*, page 8, 2010.

[4] B. Klin. Adding recursive constructs to bialgebraic semantics. *JLAP*, 60-61:259–286, 2004.

[5] B. Klin. Bialgebras for structural operational semantics: An introduction. *TCS*, 412(38):5043–5069, 2011.

[6] R. Milner. Functions as processes. *MSCS*, 2(2):119–141, 1992.

[7] M. R. Mousavi and M. A. Reniers. Congruence for structural congruences. In V. Sassone, editor, *FoSSaCS*, volume 3441 of *LNCS*, pages 47–62. Springer, 2005.

[8] D. Pous and D. Sangiorgi. Enhancements of the bisimulation proof method. In *Advanced Topics in Bisimulation and Coinduction*, pages 233–289. CUP, 2012.

[9] J. Rot, F. Bonchi, M.M. Bonsangue, D. Pous, J.J.M.M. Rutten, and A. Silva. Enhanced coalgebraic bisimulation. `http://www.liacs.nl/~jrot/up-to.pdf`.

[10] J. Rot, M. Bonsangue, and J. Rutten. Coalgebraic bisimulation-up-to. In P. van Emde Boas, F. Groen, G. Italiano, J. Nawrocki, and H. Sack, editors, *SOFSEM*, volume 7741 of *LNCS*, pages 369–381. Springer, 2013.

[11] J. Rutten. Universal coalgebra: a theory of systems. *TCS*, 249(1):3–80, 2000.

[12] D. Sangiorgi. On the bisimulation proof method. *MSCS*, 8(5):447–479, 1998.

[13] D. Sangiorgi and D. Walker. *The Pi-Calculus - a theory of mobile processes*. CUP, 2001.

[14] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291. IEEE Computer Society, 1997.

[2] In [7] it is sketched how to extend the results to the ntyft/ntyxt, which involves however a rather complicated integration of the cfsc format with the notion of stable model.

LCT-D: Proof Guided Tests for C Programs on LLVM

Olli Saarikivi and Keijo Heljanko

Department of Computer Science and Engineering,
Aalto University, School of Science,
PO Box 15400, FI-00076 Aalto, Finland
{olli.saarikivi, keijo.heljanko}@aalto.fi

Software defects can be very expensive, especially when encountered in economically critical or safety critical systems. Many of these defects can be avoided if it can be ensured that a program meets its specification. When the specification is given formally, for example with assertions embedded in the source code, automated software verification methods can be applied to determine whether a program complies to its specification.

Recently there has been much interest in combining underapproximation and overapproximation based approaches to software verification. Such a technique is employed in the DASH algorithm by Beckman et al. [1], which combines dynamic symbolic execution (DSE) [3] with counterexample guided abstraction refinement (CEGAR) [2]. DASH attempts to generate tests based on counterexamples found in the abstraction. When test generation fails the abstraction is refined to remove the counterexample. The tests can be seen as an underapproximation of the reachable states of the program under test, which DASH tries to expand to include an error. The abstraction on the other hand is an overapproximation which, if error free, also proves the program under test to be so.

The flowchart in Figure 1 provides a high-level overview of the DASH algorithm. DASH implements a modified CEGAR loop, where instead of directly checking whether a counterexample is spurious, DSE is used to generate a test that follows the path to the error in the abstraction at least one step more than in previously executed tests. When test generation fails abstraction refinement is performed to eliminate the path from the abstraction.

To explain the algorithm better we will apply DASH to the program in Figure 2. The example program takes an input, which is marked by the call to `input()`. We wish to verify that no matter what this input value is the program can not execute the error statement on line 4.

At startup DASH creates the initial abstraction from the program's control flow graph (CFG): each program location corresponds to one node in the graph, called a region, and there is a directed edge between two regions if control could flow from the first region to the second. Now each region represents all states of the program at that program location. DASH also runs one initial test on the program with a random input. Let us say this input is 41, in which case

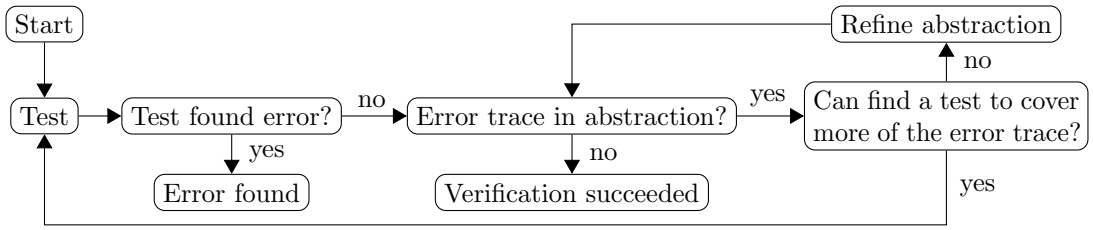
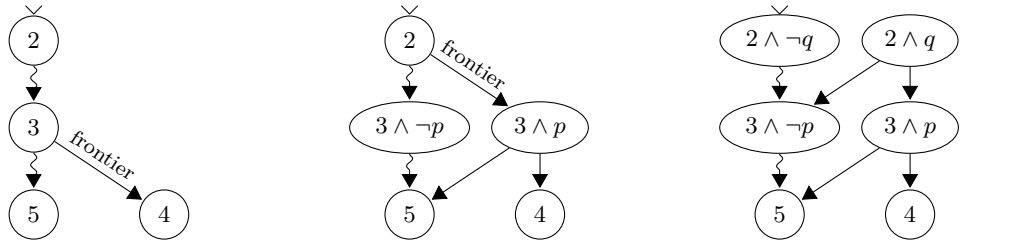


Figure 1: Flowchart for the DASH algorithm

```
int main() {                                    1
    int x = input() * 2;                        2
    if (x % 2 != 0)                             3
        error();                                4
    return 0;                                   5
}                                               6
```

Figure 2: Example program to verify with DASH



(a) After initial test run (b) After first split (c) After splitting the final frontier

Figure 3: Stages of the abstraction for the example program in Figure 2

the program does not enter the body of the `if` statement. The initial abstraction can be seen in Figure 3(a), where the path of the initial test is marked by the wavy lines.

The first iteration of the algorithm starts by finding an abstract error trace $(2, 3, 4)$. A central concept in DASH is the *frontier*, which is the edge along an abstract error trace where the first region has been visited by a test and the second one has not. Here the frontier is $(3, 4)$. DASH will now attempt to extend the frontier by using the constraints gathered from the initial test (as is done in DSE) to generate a test that would execute up to the frontier and across it. In this example the constraint would be $(x = (input * 2)) \wedge ((x \bmod 2) \neq 0)$, which is unsatisfiable. Because a test can not be generated the abstraction is refined to remove the abstract error trace. The refinement is done by splitting region 3 with a predicate p, which is such that when p is false then the execution would never proceed to region 4 from any state in region 3. In DASH such a predicate is produced by a technique similar to how weakest preconditions can be constructed. For more information see the paper by Beckman et al. [1]. In this example $((x \bmod 2) \neq 0)$ is a suitable predicate. Once the predicate has been constructed the first region at the frontier is split into versions where p is true and where p is false, which can be seen in Figure 3(b). The edge from region $(3 \wedge \neg p)$ to region 4 is eliminated, which removes this abstract error trace.

On the second iteration a new abstract error trace $(2, 3 \wedge p, 4)$ is found. Again DASH attempts to generate a test to cross the frontier (marked in Figure 3(b)), but this time the predicate p in the region $(3 \wedge p)$ is added to the constraint for test generation. However, the constraint is again unsatisfiable and the abstraction will be refined. The resulting graph from splitting the frontier with a new predicate q can be seen in Figure 3(c). Now the abstraction no longer contains a path from the initial region to the error and therefore the verification task is done.

We have implemented the DASH algorithm as a modification to the Lime Concolic Tester (LCT) [4], which is an open source dynamic symbolic execution (DSE) tool for C and Java programs. Our tool LCT-D extends the LLVM based C support in LCT.

LCT uses a client-server model to distribute test execution and constraint solving work. A testing server keeps track of the execution tree and selects which paths are to be explored next. When a client, which is an instrumented version of the program under test, connects to the

server a new path to explore is selected and the constraint corresponding to that path is sent to the client. The client calls an SMT solver with this constraint and if it is satisfiable the client executes the program with the obtained inputs. During execution the client sends details of each instruction it executes to allow the server to record constraints for generating further tests. The clients lose all state after each execution and all persistent state is stored on the testing server.

Our implementation of DASH in LCT-D follows the same general model. However, in DASH the clients (which have access to the executable program) are used in two modes: (1) to execute tests with previously solved inputs and (2) to solve constraints for generating new tests. In our tool these two modes can not be combined like they can be in normal DSE, because when solving constraints we execute the program up to the frontier to recover the concrete state of the program. This could be avoided by storing complete program states for test runs on the server, but we chose not to due to memory usage concerns. When a new set of inputs is solved on a client it is sent to the server to be used in a subsequent test execution. Currently LCT-D does not support multiple concurrent clients.

The DASH algorithm requires some way to map states visited in concrete test executions back to regions in the abstraction. The YOGI tool [5], in which the DASH algorithm was originally implemented, does this by evaluating the predicates in regions with the complete concrete states along a test execution. One of our contributions to DASH is how LCT-D infers the correct regions from control flow information combined with only the concrete values of pointers used by the program.

For more information on LCT-D and our improvements to DASH see the Master's Thesis of Olli Saarikivi [6]. The version of LCT-D used for the evaluation in the Master's Thesis is available at `http://users.ics.aalto.fi/osaariki/lctd-msc/`.

References

[1] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 3–14. ACM, 2008.

[2] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[3] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[4] Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko, and Ilkka Niemelä. LCT: An open source concolic testing tool for Java programs. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, pages 75–80, 2011.

[5] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The YOGI project: Software property checking via static analysis and testing. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181. Springer, 2009.

[6] Olli Saarikivi. Test-guided proofs for C programs on LLVM. Master's thesis, Aalto University, School of Science, Department of Information and Computer Science, 2013. `http://users.ics.aalto.fi/osaariki/msc-thesis-osaariki.pdf`.

Modeling Critical Systems with Timing Constraints in Event-B

Faezeh Siavashi[1], Marina Waldén[1], Leonidas Tsiopoulos[1] and Jüri Vain[2]

[1] Åbo Akademi University, Turku, Finland
fsiavash@abo.fi, mwalden@abo.fi, ltsiopou@abo.fi
[2] Tallinn University of Technology, Tallinn, Estonia
vain@ico.ee

1 Introduction

The complexity of safety critical systems consisting of software and hardware parts is continuously increasing. Formal methods address the issues of provably correct design offering mathematical techniques to create specifications to develop and verify safety critical systems [1]. They ensure that the implemented systems work correctly according to the defined specifications. In this paper, we study the practical aspects of applying Event-B [1] for modelling and verification of time-critical systems. Event-B has been used for developing industrial strength systems, but it lacks timing support. UPPAAL [8], on the other hand, is a model checker which has a good support for timing. In order to enrich the application areas of Event-B, we aim at extending it with timing aspects from UPPAAL. By adding timing properties to Event-B, we can guarantee provably correct timing design on the same basis as the functional correctness is ensured [3].

Event-B is based on the B-Method and is meant for refinement-based development of distributed and reactive systems where implementation details are added to design specifications in a stepwise manner. The system model is extended with new variables and assignments, and new conditions, e.g. stronger guards and invariants. Event-B comes with the Rodin tool, that provides automatic and interactive discharging of proof obligations [5]. UPPAAL is a model checker with extended timed automata called UPPAAL Timed Automata (UPTA)[2].

Our main contribution is that we exploit the patterns for modeling and refinement of timing properties within UPPAAL and transform these patterns to patterns in Event-B. Hence, we are able to verify that the refined timing specification combined with refined functionality together satisfy the more abstract specification [6]. Our work is exemplified by a case study provided by Danfoss Power Electronics, which was part of the EU-project RECOMP (2010-2013) [4]. The case study is available in detail in [7].

2 Model transformation from UPPAAL to Event-B

We model the timing properties of the system in UPPAAL. These models are then transformed to Event-B as follows: (1) Each UPPAAL model location is mapped to a state of Event-B. (2) Each transition between locations in the UPPAAL model is mapped to an event in Event-B. (3) The abstract clock in UPPAAL is mapped to an event in Event-B. (4) The invariants and guards in UPPAAL are modeled to guards in Event-B. (5) The declarations in the UPPAAL model is mapped to invariants and axioms in Event-B, according to the data types of the parameters in UPPAAL.

Real-time systems contain a variety of patterns for timing constraints. In this work, we focus on the two most important and common timing constraints and their refinement patterns: Delay

and Deadline.

The delay pattern. In the abstract level delay is modelled as an integer type counter that increases by one at each clock cycle of the system. The refined clock is modeled with the system clock that progresses a certain number of ticks within each abstract clock cycle.

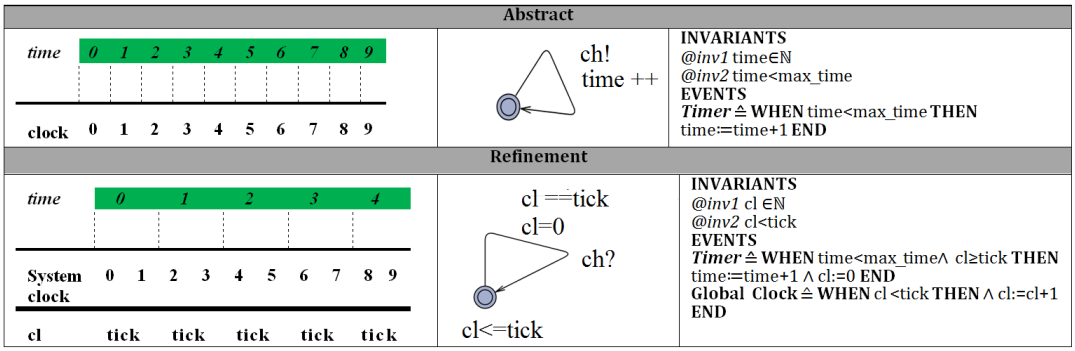


Figure 1: Delay modeling and its refinement in UPPAAL and Event-B models

Delay is modeled in UPPAAL by two UPTA models for the abstract and the refined clock that are translated to models in Event-B (see Figure 1). In the refined Event-B model we introduced a new timing variable 'cl' with a new event **Global_Clock** to model the global timer of the system. The guard of event **Timer** is refined to consider this global timer. We note that the event **Timer** is only modified by strengthening its guard and adding assignments to the new timer variable preserving the old behavior and the invariant. The new event **Global_Clock** assigns the new variable 'cl' while preserving the invariant. Moreover, the upper limit of 'cl' in the model guarantees the non-divergence of the event. Hence, the refined model is a correct refinement of the abstract model.

Nested Time Interval (Deadline pattern). We have two transitions $e1$ and $e2$. They occur within a specific time span ($t1 \leq e1 < e2 \leq t2$) that is refined to ($ref\_t1 \leq e1 < e2 < ref\_t2$). In the abstract UPTA model, the deadline of $e1$ is modeled by an invariant ($cl < t2$) and a guard ($cl >= t1$) and in the refinement the deadlines are shorter (see Figure 2). In Event-B, this is modeled by three events **e1_Change**, **e2_Change** and **Timer**. Due to the decreased time intervals of the events, the guards are strengthened. The correctness of the refined deadline interval is ensured by the events **e1_Change** and **e2_Change** preserving the invariant. Since the action part of the events are not changed, it trivially guarantees that the behavior of the model is not changed.

Case Study. The Delay and Deadline patterns have been applied on an industrial case study, where a frequency converter with two reset buttons are connected to a pair of redundant processors via a safety module. The reset buttons shutdown the converter whenever there is a difference between power cycles of the motor. There are two different safety functions called SafeStop1 (SS1) and SafeTorqueOff (STO) that can be activated. Whenever Emergency Shutdown (ES) button is pushed for some amount of time (delay), the system will be reset. The reaction is based on two-step reset: first the SS1 signal will be activated and then within a certain time (deadline) the STO signal will be activated to shut down the system. The delay and deadline patterns introduced above were used to refine the ES delay and the deadline time for the STO [7].

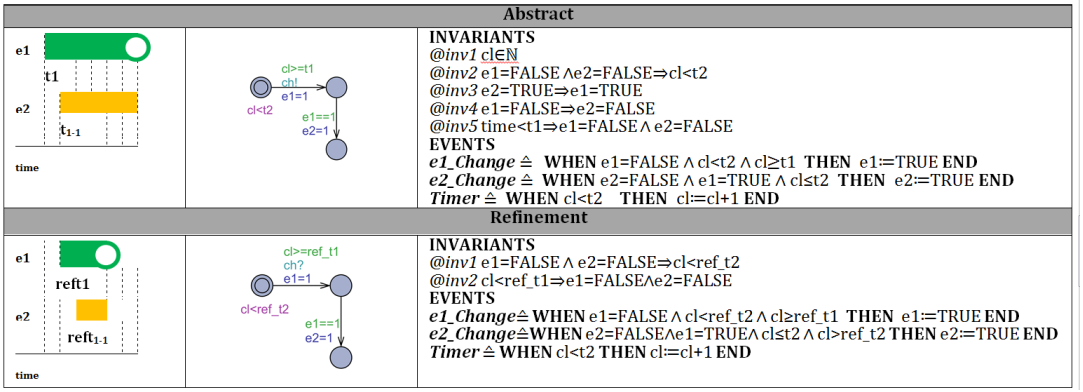


Figure 2: Nested time intervals (deadline) modeling and refinement in UPPAAL and Event-B

3 Conclusion

The main problem is defining a clock in Event-B, since it is not similar to a real clock, which models continuous time with identical time slots for each clock cycle. In Event-B, we have defined a clock as a discrete time element which does not necessarily increase continuously. In addition, the clock cycles in a discrete timer do not have the same duration. It is possible that some of the clock cycles take shorter time while others take longer time. This is because of the nature of the Event-B language. The progress of time is dependent on the events in the model rather than on a reference clock that is running with its own rate. In case more than one event are enabled at a time, Event-B can give priority to an event suspending the clock event. Moreover, an event which is enabled will not necessarily be executed.

Modeling time in Event-B mostly covers properties and temporal relations of events. It ensures that if the deadline for the execution of an event is passed and the event missed the deadline, it cannot be executed. If any of the events misses its deadline, then the reliability of the system is not either assured, since the reliability of real-time systems often depends on the response time. Guards and invariants of the model guarantee that if the timer does not prohibit the executions of the other events before the deadline passes, then the events will occur in correct time and order.

References

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

[2] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. Springer, 2004.

[3] D. Cansell, D. Méry, and J. Rehm. Time constraint patterns for event b development. In the Proc. of *Formal Specification and Development in B*, LNCS 4355. Springer, 2006.

[4] RECOMP. http://atcproyectos.ugr.es/recomp/, last access 10/11/13.

[5] RODIN. Event-B and the Rodin platform. http://www.event-b.org/, last access 10/11/13.

[6] M. R. Sarshogh and M. Butler. Specification and refinement of discrete timing properties in Event-B. In the Proc. of AVoCS 2011. September 2011.

[7] F. Siavashi. Modelling critical systems with time constraints in Event-B. Master's thesis, Åbo Akademi University, 2012.

[8] UPPAAL. http://www.uppaal.org, last access 10/11/12.

Verification of Graph-based Model Transformations Using Alloy

Xiaoliang Wang[1] and Yngve Lamo[1]

Bergen University College, Norway
{xwa,yla}@hib.no

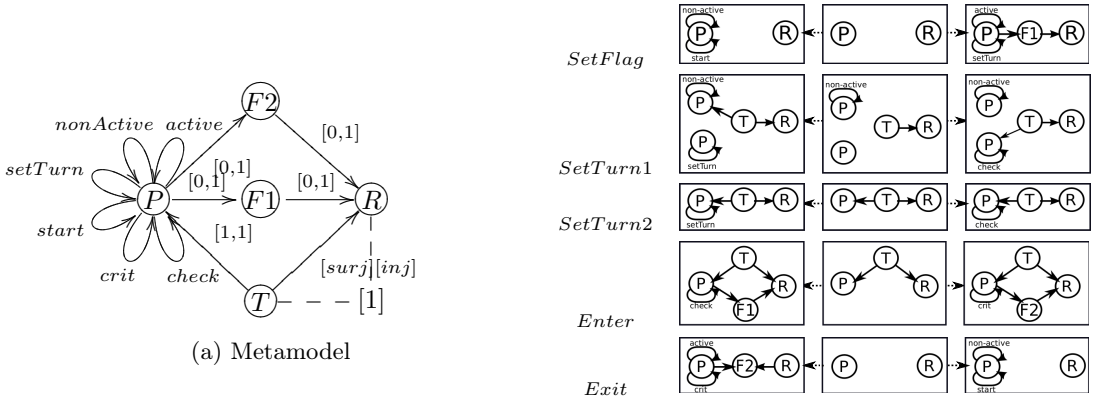
1 Introduction

In model driven engineering (MDE), models are considered the basis for software development. They are used to specify the domain under study, to generate program code and for documentation purposes etc. Ideally, a model in the next development phase can be automatically generated from a model used in the previous phase by model transformations. Such automation makes MDE appealing by offering more consistent software and higher productivity. However, validation of model transformations should be ensured. Without validation, errors in some transformations is transferred to the following phase, which may result in erroneous software. Usually, a model transformation is executed by applying model transformation rules on a model. A model transformation system consists of a set of such rules. Our work aims to verify if a model transformation system is correct w.r.t. conformance, i.e. for each valid source model is the target model obtained after the transformation still valid. We focus on graph-based model transformations [3] and present a bounded verification approach based on first order logic (FOL). The idea is to translate a model transformation system into a relational logic specification. Then we use the Alloy model analyzer[1], to check if any invalid target model are created by the transformation. To illustrate our approach, we run an example in Diagram Predicate Framework (DPF) [4], a framework which provides diagrammatic modelling and model transformations based on graph transformations. The example will be expressed in relational logic before the Alloy Analyzer [1] will be used to verify the system. Note that the approach can be proceeded automatically in DPF.

2 Verification Example

A model transformation system [3] consists of a metamodel $\mathcal{MM}$[1] and a set of model transformation rules $\{r : L \xleftarrow{\varphi} K \xrightarrow{\psi} R\}$. L, K, R are the left-hand side, the gluing graph and the right-hand side. The two morphisms φ and ψ are injective. L and R are typed by $\mathcal{MM}$, but is not necessary a valid instance of $\mathcal{MM}$. Figure 1 shows a variant of Dijkstra's algorithm for mutual exclusion [2] presented as a model transformation system in DPF. The algorithm ensures that a critical resource is accessed exclusively by one process each time. In DPF, the structural syntax is specified by a directed graph. R is the resource which processes P access. T tells which process is currently accessing R. The flags $\{F1, F2\}$ and the states $\{nonActive, active, start, crit, check, setTurn\}$ are used to control how to access R. $T \rightarrow P$ means that the P is eligible to access R. A reflexive arrow on P labeled with one of the six states means that the process is at such a state. An arrow from P to one of the two flags means

[1] For exogenous transformations, an integrated metamodel which interrelates the source and target metamodel can be construct. Then the integrated metamodel can be used as the metamodel of the model transformation system. Examples can be found in [3].



(a) Metamodel

(b) Transformation rules

Figure 1: Dijkstra Mutual Exclusive Model Transformation System

that P is marked with such a flag. Here, constraints are specified by diagrammatic predicates on part of the graph, where predicates are denoted by $[PredicateName]$. For example, each flag may have at most one arrow to R, which is ensured by the multiplicity constraint $[0, 1]$ on the respective arrows.

Based on the diagrammatic modelling framework, DPF also provides a framework to specify constraint-aware model transformations[5], which means that the transformation rules may contain constraints. However in this paper we only consider metamodel constraints. In the example, the metamodel is shown in Figure 1a while Figure 1b shows the model transformation rules. Rule $SetFlag$ requests access to R. Rule $SetTurn1$ and $SetTurn2$ assign T to one process depending on the context. Rule $Enter$ lets the eligible process access R, while rule $Exit$ finishes accessing R.

3 Verification Approach

We can use existing tools to specify model transformation systems, but most tools have no verification mechanism for model transformations. In this section we will propose a bounded verification approach, the idea is to translate a model transformation system to a relational logic specification. Each component; metamodel (including structure and constraints) and model transformation rules, can be encoded in the logic. In DPF the structure of a metamodel is a graph, hence we can use functions and constraints to express the graph. Those functions and constraints represent all the possible model instances typed by the graph. For example, nodes and edges in metamodels are encoded as Alloy signatures:

```
sig <NodeType>{}                    sig <EdgeType>{src, trg: one <NodeType>}
```

Here, an explicit constraint that each edge should have one source node and one target node, is translated as one in this signature. Metamodel constraints further restricts the instances of the metamodel. It should be noticed that, since our approach is based on FOL, constraints are restricted to those which can be expressed in FOL. For example, the constraint $[surj]$ on Arrow $T \rightarrow F1$ can be expressed as a $Fact$ in Alloy as follows:

```
fact{all n : SV_F1 | one e : SE_PF1 | e.trg=n}
```

In this work, we focus on graph-based model transformations using a classical double-pushout (DPO) approach [3]. For each transformation, a source model is transformed into a target model according to a model transformation rule. In the source model, the elements matched by the rule are deleted or transformed into target elements while others are unchanged. In the target model, the elements are added or transformed from source elements while others are unchanged. In this way, a model transformation can be viewed as a relation between the source and the target, which can also be expressed in FOL.

After we have encoded the model transformation system into a FOL specification, we can verify the system. In practice, we check if an invalid target model is created from a valid source model. If such an invalid target model is found, we can assert that the system is not correct w.r.t. conformance, otherwise, the system is assured correct. Note that the counterexample can help the designer to redesign the system.

Since we check if any invalid target model is transformed from some valid source model, constraints are handled differently depending on if they belong to the source or the target. Source constraints are translated as $fact$ and target constraints are translated into *check*. Then we use the Alloy Analyzer to verify the specification. We do not check all the rules simultaneously. A transformation may involve several rules. But the Parallelism Theorem [3] states that a transformation is equal to a sequential application of the rules. This enables us to check rules one by one. Besides, constraints are also checked one by one for simplicity. Besides, if we put all the target constraints together, it is not easy to analyse the counterexample when a system is not correct. The Alloy Analyzer performs check over a user-defined finite scopes. It means the approach can only verify a system in some finite scopes. The verification shows that the example is not correct. But adding some constraints on the source model and a Negative Application Condition (NAC) on rule $setFlag$ makes the system correct.

4 Conclusion and Future work

We have presented a bounded verification approach of graph-based model transformations. An example in DPF is given to illustrate the approach. The approach verifies a transformation system's correctness w.r.t. conformance by checking that each transformation from a valid source model can create a valid target model. However, for systems which have transformations creating invalid target models, correctness cannot be verified here. We will consider this in the future. Furthermore, since the Alloy Analyzer performs check over a user-defined scope, the approach is incomplete in that it cannot check the instances out of the scope. Moreover the limit of the scope which can be handled with the approach is not clear, this should be explored.

References

[1] Alloy. *Project Web Site*. `http://alloy.mit.edu/community/`.

[2] Edsger W Dijkstra. Solution of a problem in concurrent programming control. In *Pioneers and Their Contributions to Software Engineering*. 2001.

[3] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. Springer. 2006.

[4] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A diagrammatic formalisation of mof-based modelling languages. *Objects, Components, Models and Patterns*, 2009.

[5] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A formal approach to the specification and transformation of constraints in mde. *Journal of Logic and Algebraic Programming*, 81(4), 2012.