IFM2005 doctoral symposium on integrated formal methods, Eindhoven, The Netherlands, November 29, 2005

IFM2005
Doctoral Symposium on Integrated Formal Methods

Judi Romijn
Graeme Smith
Jaco van de Pol
(editors)

The IFM2005 doctoral symposium on integrated formal methods is sponsored by IPA, the institute for Programming Research and Algorithmics.

Table of Contents

Preface

The conference on Integrated Formal Methods is held for the fifth time, this time at the Technische Universiteit Eindhoven (the Netherlands). The conference addresses the integration of formal methods (semantics, methodologies, and/or verification algorithms), and attempts to provide meaningful and effective answers to questions regarding inter-model consistency, completeness and correctness of implementations.

The first IFM doctoral symposium was organised in 2004 at the University of Canterbury (Kent, UK) as an event preceding the main IFM2004 conference. This was clearly appreciated as quite a few Ph.D. students participated. There are not so many opportunities in this research area for Ph.D. students to present their work outside a regular conference setting, and without being forced to write full-fledged papers. We felt that this initiative deserved to be continued, and organized a doctoral symposium for IFM2005 as well, taking place on Tuesday, November 29, 2005.

The IFM2005 doctoral symposium has attracted no fewer than 20 submissions, and we were able to invite 13 quality presentations, from the Netherlands as well as surrounding countries such as the United Kingdom, Germany and France, and even the Czech Republic. This proceedings contains the abstracts of presentations on such diverse subjects as semantical integration, issues in theorem proving, timed and hybrid approaches, and the combination of formal methods with cryptography. Undoubtedly, these promising young researchers will contribute to a lively symposium, and will benefit themselves from the experience of presenting their work in public, getting feedback from their colleagues and attending the invited tutorial and the main conference.

We thank the research school IPA for sponsoring the doctoral symposium.

Judi Romijn, Graeme Smith, Jaco van de Pol

November 2005

Building Verification Condition Generators by Compositional Extension

A. J. van Leeuwen
Department of Information and Computing Sciences, Utrecht Univ.

October 13, 2005

Abstract

Current mechanizations of programming logics are often in the form of verification condition generators. These frontends to a prover translate a program and assertions into conditions that when proven state that the program fulfils its assertions. Traditional verification condition generators are monolithic encapsulations of a programming language's semantics. This makes it hard to build such verification generators when designing a new language, or when extending a language. Therefore we propose a more compositional method of building verification condition generators, using ideas from monadic denotational semantics and from generic programming. Our technique allows us to extend an existing verification condition generator to handle new language constructs, but also to add extensions at another level, such as an ability to generate validation traces. Furthermore, it allows us to weaken the logic, to do light weight verification and possibly making the verification conditions decidable. We explain the technique through an example, extending a simple while language with a construct for exception handling. This construct not only needs an extension to the logic, but also a change of its structure.

1 Introduction

Development and maintenance of the implementation of realistic programming logics is known to be hard. There are few methods of dealing with the changes that are incurred by changing the language or the addition of features. Implementing these changes is a dangerous and error prone process that can easily introduce inconsistencies in the logic.

Logics underlying imperative languages are usually syntax driven and implemented as a recursive function over the target program [5]. While this is a straightforward method of implementing these logics, it leads to monolithic programs that cannot be altered or extended easily.

We demonstrate a technique that enables us to change and extend the implementation of a logic in a modular way, that is without directly tampering with the code of the existing implementation of a logic. Our approach has a number of advantages. First and foremost is the advantage that it is safer. Second, it allows engaging and disengaging alterations at will. Thus we can always fall back on the existing implementation. Third, it enables us to easily create a set of related partial logics, each of which can be used separately for light weight verification.

Our technique uses ideas from generic programming and monadic denotational semantics to represent a syntax driven logic. Generic programming abstracts from datatypes, and these techniques can thus be used to abstract from the actual implementation of the abstract syntax used to represent programs. The main use we make is that of a generic fold, which we assume to be easily specified. This allows us to decouple the recursion scheme from the calculation of the semantics.

Monadic denotational semantics has been introduced as a method of separating concerns in denotational semantics [11, 1]. The key idea is to choose an abstraction for the domain of the semantics such that different computations in that domain can be combined in one standard fashion. By relying on that combination method, a certain independence of the domain results.

$$
\begin{aligned}
\mathit{Stmt} \rightarrow\ & \mathit{Variable} := \mathit{Expr} \\
\mid\ & \textbf{if } \mathit{Expr} \textbf{ then } \{\mathit{Stmt}\} \textbf{ else } \{\mathit{Stmt}\} \\
\mid\ & \textbf{inv } \mathit{Expr} \textbf{ while } \mathit{Expr} \textbf{ do } \{\mathit{Stmt}\} \\
\mid\ & \mathit{Stmt}; \mathit{Stmt}
\end{aligned}
$$

Figure 1: Simple imperative language L

$$
\begin{array}{l}
\text{pre } (x\ :=\ e)\ q\ =\ q[e/x] \\
\text{pre } (S_1;\ S_2)\ q\ =\ \text{pre } (S_1)\ (\text{pre } (S_2)\ q) \\
\text{pre } (\textbf{if } g \textbf{ then } \{S_1\} \textbf{ else } \{S_2\})\ q\ =\ \mathit{if}\ g\ \mathit{then}\ \text{pre } (S_1)\ q\ \mathit{else}\ \text{pre } (S_2)\ q \\
\dfrac{p\ =\ \text{pre } (S)\ i \quad i \wedge \neg g \Rightarrow q \quad i \wedge g \Rightarrow p}{\text{pre } (\textbf{inv } i \textbf{ while } g \textbf{ do } \{S\})\ q\ =\ i}
\end{array}
$$

Figure 2: Derivation rules for pre

Thus, given a monadic domain, we can hide those aspects that do not affect a specific part of the logic. This allows us to keep the code of the base logic unchanged despite adding to the underlying structure, which normally would affect the implementation of each rule in the logic.

2 Implementing a logic

Consider the simple imperative language L in shown in figure 1. For simplicity, we will assume that all statements terminate. The construct **inv** i **while** g **do** S is a simple while loop where i is a candidate invariant specified by the programmer.

We will specify a verification condition generator for the logic of this language in terms of a predicate tranformer logic [3]. A predicate transformer is a function that takes a predicate and a statement in a language, and returns a predicate transformed according to the semantics of that statement. In the most wellknown instantion, this function takes postconditions into preconditions, such that $p = \text{pre } P\ q \Rightarrow \{p\}P\{q\}$ where $\{p\}P\{q\}$ is a Hoare-triple stating that pre- and postcondition p and q are valid for statement P. Figure 2 shows rules that specify how the value of pre can be computed. Note that the value is only valid if the additional conditions that occur in the calculation for the while rule also hold.

This specification is syntax driven: for each production rule in the syntax, there is one rule specifying how to calculate pre. Therefore, calculating the value of pre $(S)\ q$ given a statement S and a post-condition q is a matter of recursing over the structure of S. Some rules emit extra verification conditions that should also be collected, thereby resulting in the term *verification condition generator* for the implementation of pre. The straightforward implementation of such a verification condition generator would compute both the precondition and collect the verification conditions. Let us name the function *pvcg* Given a program s, a post-condition q and an initially empty list of verification conditions, $\mathit{pvcg}\ s\ q\ [\,]$ woulrd return a tuple (p, vcs) where vcs is a list of verification conditions and p is pre $(s)\ q$. Its type would therefore be $\mathit{pvcg} :: \mathit{Stmt} \rightarrow \mathit{Expr} \rightarrow [\mathit{Expr}] \rightarrow (\mathit{Expr}, [\mathit{Expr}])$.

While straightforward, such code is not easily changed. However, as it is syntax derived, the way to implement it as a fold over the abstract syntax can be easily seen: it involves making a separate function for each nonterminal in the abstract syntax, and passing those functions as arguments to the fold.

This decouples the recursion from the calculation of the precondition and the other verification conditions. Unfortunately the code then still strongly depends on the exact structure of the domain. We can improve on that.

$$\begin{aligned}
&\mathit{pvcg\_Assign}\ x\ e && = \lambda q \rightarrow \mathit{return}\ (\mathit{subst}\ (x, e))\\
&\mathit{pvcg\_Seq}\ \mathit{p\_s1}\ \mathit{p\_s2} && = \lambda q \rightarrow \mathit{p\_s2}\ q \ggg\!= \lambda p' \rightarrow \mathit{p\_s1}\ p' \ggg\!= \lambda p \rightarrow \mathit{return}\ p\\
&\mathit{pvcg\_IfThenElse}\ g\ \mathit{s1}\ \mathit{s2} && = \quad \text{-- similar to others}\\
&\mathit{pvcg\_InvWhile}\ i\ g\ \mathit{p\_body} && = \lambda q \rightarrow \mathit{p\_body}\ i \ggg\!= \lambda p \rightarrow\\
& && \qquad \mathit{record}\ (i \wedge \neg\ g \rightarrow q) \ggg\!= \lambda\_ \rightarrow\\
& && \qquad \mathit{record}\ (i \wedge g \quad \rightarrow p) \ggg\!= \lambda\_ \rightarrow\\
& && \qquad \mathit{return}\ i
\end{aligned}$$

$$\mathit{pvcg} = \mathit{foldStmt}\ (\mathit{pvcg\_Assign}, \mathit{pvcg\_Seq}, \mathit{pvcg\_IfThenElse}, \mathit{pvcg\_InvWhile})$$

Figure 3: Folded monadic implementation of a vcg for L

$$\begin{aligned}
&\mathit{pvcg\_Raise}' && = \lambda q \rightarrow \mathit{return}\ \mathit{false}\\
&\mathit{pvcg\_TryCatch}'\ \mathit{p\_s1}\ \mathit{p\_s2} && = \lambda q \rightarrow \mathit{p\_s1}\ q
\end{aligned}$$

$$\begin{aligned}
&\mathit{pvcg} = \mathit{foldStmt}\ (\mathit{pvcg\_Assign}, \mathit{pvcg\_Seq}, \mathit{pvcg\_IfThenElse}, \mathit{pvcg\_InvWhile},\\
&\quad \mathit{pvcg\_Raise}', \mathit{pvcg\_TryCatch}')
\end{aligned}$$

Figure 4: Initial extension of pvcg from L to L_1

Looking at the type of the implementation, $\mathit{Stmt} \rightarrow \mathit{Expr} \rightarrow [\mathit{Expr}] \rightarrow (\mathit{Expr}, [\mathit{Expr}])$, we see that it takes a statement and an expression and returns a new expression, if we ignore the fact that we also thread the collection of other verification conditions through the calculation. However, using monads, we can abstract from that threading.

A monad can be taken to represent a computation over a value, such as collecting conditions alongside a predicate. We have two functions for monads, *return* to inject values into computations, and $\ggg\!=$ to sequentially combine a computation over a value with a function taking that value and returning a new computation.

We can state the implementation's type as $\mathit{Stmt} \rightarrow \mathit{Expr} \rightarrow m\ \mathit{Expr}$, where the monad m abstractly represents the act of collecting the side conditions. Using a fold and monads, the code for the verification condition generator would then look somewhat like that in figure 3. Note that we assume a function *record* that will record verification conditions in values of type $m\ ()$, and that we use the abstract sequential combination of computations as opposed to explicit threading of values.

3 Modifying a logic

Modifying a straightforward implementation as a recursive function involves changing the entire function. Modifying the implementation in 3 involves changing only those functions affected. Furthermore, as we abstract from the structure of the domain in 3, it is possible to change that independently.

Suppose we add exception handling to our language, adding **raise** and **try** $\{\mathit{Stmt}\}$ **catch** $\{\mathit{Stmt}\}$. We will need to write new rules to handle the new constructs. We assume that expressions do not in themselves raise exceptions. Now, if we postulate that no valid program may execute **raise**, and simply take the precondition of a **try** $\{\mathit{s1}\}$ **catch** $\{\mathit{s2}\}$ block to be that of *s1*, we get the simple implementation in figure 4.

This code, while reusing the old code, is not entirely satisfactory. A more satisfactory implementation would actually handle the exceptions. This requires making a distinction between the postcondition in the normal case, and the postcondition in the exceptional case. Our technique can handle that, with ease! Assume our computational monad m not only records verification conditions, but also stores the current postcondition for the exceptional case, with functions *getPostE* and *setPostE* to access that postcondition. Then we can write our verification condition generator as in figure 5. Note that adding the possibility for expression to raise an exception would involve adding invocations to *getPostE* in the appropriate places.

```
pvcg_Raise               = λq → getPostE
pvcg_TryCatch p_s1 p_s2 = λq → getPostE ≫= λq′ →    p_s2 q ≫= λp′ →
                                setPostE p′ ≫= λ_ → p_s1 q ≫= λp →
                                setPostE q′ ≫= λ_ → return p
pvcg = foldStmt (pvcg_Assign, pvcg_Seq, pvcg_IfThenElse, pvcg_InvWhile,
  pvcg_Raise, pvcg_TryCatch)
```

Figure 5: Full pvcg for L_1 using code from L

Note that we have left the method of specifying the monad encapsulating the domain implicit. One way of specifying this monad is by directly choosing one. While this works, it involves replacing the entire monad upon changes. Much nicer would be the ability to just add aspects to the existing monad. Traditional approaches of combining monads are the use of distributive laws over monads [7] or the use of monad transformers [4, 10]. We have chosen to use another approach, that of monadic coproducts [9].

A monad forms a computational layer over a certain value. If we want to combine computational actions from different monads, we can put multiple computational layers over a value. However, when choosing one particular layering of monads for the combined monad, in general one loses the $\gg\!=$ function. In a monadic coproduct we do not chose one particular layering but rather all possible layerings of two monads. Unfortunately, this denotes a largely redundant computational structure, as one is interested in computational actions rather than in layers of possible computational actions. The trick is to identify the actual computational actions, and quotient the coproduct's structure so that each actual computational action is only represented once in the coproduct.

To construct a monadic coproduct, one needs finitary layered monads, that is monads for which the behaviour on infinite objects is determined by its behaviour on finite objects, and for which it is possible to determine if the monadic value is in the range of the *return* function. The latter requirement allows us to distinguish between layers in which an action has been performed and layers which may be safely ignored, and therefore allows us to actually calculate the required quotient. Fortunately, many practical monads have these properties.

4 Research direction

Given this implementation method for verification condition generators, a number of research paths remain. Not only do we want a more compositional way of implementing a verification condition generator, we also want to formally verify said generator, as for example Homeier and Martin described [5]. One hypothesis we have is that our approach makes it possible to not only reuse exsiting code, but also to reuse proofs so that proving e.g. soundness for a changed verification condition generator need not be done from scratch.

Another research path is that of combining this method of specifying a logic with the existing methods of building monadic interpreters and compilers, so that when designing a language one can declaratively specify syntax and semantics, and get a complete implementation of the language, including a proofsystem, for free.

5 Related Work

The usefulness of monads for programming language semantics was already realized by Moggi and Cenciarelli [11, 1], and then expanded upon by Espinosa, Liang and Hudak [4, 8, **?**]. Monads were applied to proving properties over programs by Jacobs and Poll [6] and more recently by Schröder and Mossakowski [13]. Our approach differs from these in that instead of specifying a logic that is independent of the underlying monad, we use the monad to implement a logic, and instead of developing one single monad that captures the entire semantics of a single programming language, we intend to combine semantical aspects into such a monad.

The use of folds in our approach is reminiscent of attribute grammars [12]. For our purposes we would need a method of introspection on the grammar rules, so that we can perform the modifications that we described. The first class attribute grammars by De Moor, Backhouse and Swierstra [2] may well fit the bill. The methods of composing these grammars also suggest another approach to specifying the semantics.

6 Conclusion

We have shown a technique to deal with changes of and extensions to the implementation of a verification condition generator. This technique provides for a method of modular development of the verification condition generator in the face of changing requirements. Preliminary experiments have shown that the technique may work. We believe it is worth further investigation.

References

[1] P. Cenciarelli and E. Moggi. A syntactic approach to modularity in denotational semantics. In *CTCS 5*. CWI, 1993.

[2] O. de Moor, K. Backhouse, and S. D. Swierstra. First class attribute grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. Special Issue: Attribute grammars and Their Applications.

[3] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics.* Texts and Monographs in Computer Science. Springer Verlag, Berlin, 1990.

[4] D. Espinosa. *Semantic Lego.* PhD thesis, Columbia University, 1995.

[5] P. V. Homeier and D. F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In *TPHOLS 1994*, number 859 in LNCS, pages 269–284, Malta, 1994. Springer-Verlag.

[6] B. Jacobs and E. Poll. A monad for basic java semantics. In T. Rus, editor, *AMAST 2000*, number 1816 in LNCS, pages 150–165. Springer, 2000.

[7] D. J. King and P. Wadler. Combining monads. In J. Launchbury and P. M. Sansom, editors, *Proceedings of the Glasgow Workshop on Functional Programming*, Workshops in Computing Series, Glasgow, July 1992. Springer Verlag.

[8] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL'05*, pages 333–343, New York, NY, USA, 1995. ACM Press.

[9] C. Lüth and N. Ghani. Composing monads using coproducts. In *ICFP'02*, pages 133–144, New York, NY, USA, 2002. ACM Press.

[10] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, University of Edinburgh, 1989.

[11] E. Moggi. Computational lambda-calculus and monads. In *LICS'89*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[12] J. Paakki. Attribute grammar paradigms — a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.

[13] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in hascasl. *Journal of Logic and Computation*, 14(4):571–619, 2004.

Semiring Neighbours: An Algebraic Embedding and Extension of Neighbourhood Logic

Peter Höfner

Institut für Informatik, Universität Augsburg
D-86135 Augsburg, Germany
hoefner@informatik.uni-augsburg.de

Abstract. In 1996 Zhou and Hansen proposed a first-order interval logic called *Neighbourhood Logic* (NL) for specifying liveness and fairness of computing systems and also defining notions of real analysis in terms of expanding modalities. After that, Roy and Zhou presented a sound and relatively complete Duration Calculus as an extension of NL.
We present an embedding of NL into an idempotent semiring of intervals. This embedding allows us to extend NL from single intervals to sets of intervals as well as to extend the approach to arbitrary idempotent semirings. We show that most of the required properties follow directly from Galois connections, hence we get the properties for free. As one important result we get that some of the axioms which were postulated for NL can be dropped since they are theorems in our generalisation. At the end of the paper we shortly present some possible applications for neighbours beyond intervals. Here we discuss for example reachability in graphs and applications for hybrid systems.

1 Introduction and related work

Chop-based interval temporal logics, such as ITL [4] and IL [2] are useful for the specification and verification of safety properties of real-time systems. In these logics, one can easily express a lot of properties such as

"if ϕ holds for an interval, then there is a subinterval where ψ holds“.

As it is shown in [13], these logics cannot express all desired properties. E.g., (unbounded) liveness properties such as

"eventually there is an interval where ϕ holds“

is not expressible in these logics. Surprisingly, these logics cannot even express state transitions. That is why in Chapter 9 of [13] extra atomic formulas are introduced. As it is shown there the reason is that the modality *chop* $\frown$, is a *contracting* modality, in the sense that the truth value of $\phi\frown\psi$ on $[b,e]$ only depends on subintervals of $[b,e]$:

$\phi\frown\psi$ holds on $[b,e]$ iff
there exists $m \in [b,e]$ such that ϕ holds on $[b,m]$ and ψ holds on $[m,e]$.

Hence Zhou and Hansen proposed a first-order interval logic called *Neighbourhood Logic* (NL) in 1996 [14]. This first-order logic was proposed for specifying liveness and fairness of computing systems and also defining notions of real analysis in terms of expanding modalities. In 1997 Roy and Zhou presented a sound and relatively complete Duration Calculus as an extension of NL [11]. They had already shown that the basic unary interval modalities of [5] and the three binary interval modalities (C, T and D) of [12] could be defined in NL.

In this paper, we present an embedding of NL into the semiring of intervals presented e.g. in [8]. This embedding allows us to extend NL from single intervals to sets of intervals as well as to extend the approach to arbitrary idempotent semirings. Because of work done in [14] it is also an extension of [5] and [12]. In Section 4 we show that most of the required properties follow directly from Galois connections, hence we get the properties for free. As one important result we get that some of the axioms which were postulated for NL can be dropped since they are theorems in our generalisation. At the end of the paper we briefly present some possible interpretations of neighbours in other models. Here we discuss for example reachability in graphs and applications for hybrid systems. Due to lack of space all proofs are skipped. They can be found in [7].

2 About Neighbourhood Logic

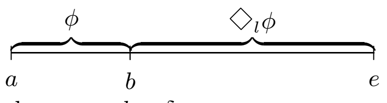
In [14] Zhou and Hansen introduce *left* and *right* neighbourhoods as primitive intervals to define other unary and binary modalities of intervals in a first-order logic. For this, we need intervals as carrier sets. That is why we define *intervals* over a poset of *timepoints* in the usual way as

$$[b,e] \overset{\text{def}}{=} \{x \,:\, b \le x \le e\}\ ,\ \text{where } b \le e,$$

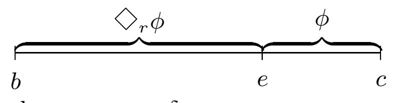
$b, e, x \in \mathbb{T}\text{ime}$ and $(\mathbb{T}\text{ime}, +, 0)$ is a monoid. Furthermore, we postulate a subtraction $-$ on $\mathbb{T}$ime satisfying for any interval $[b,e]$ the equations $e - b \ge 0$ and $e - b = 0 \Leftrightarrow e = b$. Hence, it is possible to calculate the *length* l of the interval $[b,e]$ as $e - b$. Additionally, $\mathbb{T}$ime has to be cancellative w.r.t. $+$. E.g. one can use $\mathbb{R}$, the set of real numbers, as $\mathbb{T}$ime.
The two proposed simple expanding modalities $\Diamond_l\phi$ and $\Diamond_r\phi$ are defined as follows:

$\Diamond_l\phi$ holds on $[b,e]$ iff there exists $\delta \ge 0$ such that ϕ holds on $[b-\delta, b]$,
$\Diamond_r\phi$ holds on $[b,e]$ iff there exists $\delta \ge 0$ such that ϕ holds on $[e, e+\delta]$,

where ϕ is a *formula* of NL, which is either true or false.[1] With $\Diamond_r(\Diamond_l)$ one can reach the *left* (*right*) *neighbourhood* of the beginning (ending) point of an interval:



where $a = b - \delta$



where $c = e + \delta$

[1] The exact definition of the syntax can be found in, e.g., [14, 7].

In contrast to the chop operator the neighbourhood modalities are *expanding* modalities, i.e., they are not contracting operators. Thus $\Diamond_l$ and $\Diamond_r$ depends not only on subintervals of an interval $[b, e]$, but also on intervals "outside". In [14] it is shown that the modalities of [5] and [12] as well as the chop operator can be expressed by the neighbourhood modalities.

3 Embedding Neighbourhood Logic into semirings

First, we repeat the basic definitions of semirings and related algebraic structures and operators. More details about semirings, domain semirings, etc. can be found in [6, 1, 3].

A *semiring* is a quintuple $(S, +, \cdot, 0, 1)$ such that $(S, +, 0)$ is a commutative monoid and $(S, \cdot, 1)$ is a monoid such that $\cdot$ is distributive over $+$ and S is *strict*, i.e., $0 \cdot a = 0 = a \cdot 0$. The semiring is *idempotent* if $+$ is, i.e., $a + a = a$. On idempotent semirings the relation $a \leq b \overset{\text{def}}{\Leftrightarrow} a + b = b$ is a partial order, called the *natural order* on S. The definition implies that 0 is the least element and $+$ and $\cdot$ are isotone with respect to $\leq$. If S has a greatest element we denote it by $\top$. An important semiring is REL, the algebra of binary relations over a set under relational composition.

A *test semiring* is a pair $(S, \mathsf{test}(S))$, where S is an idempotent semiring and $\mathsf{test}(S) \subseteq [0, 1]$ is a Boolean subalgebra of the interval $[0, 1]$ of S such that $0, 1 \in \mathsf{test}(S)$ and join and meet in $\mathsf{test}(S)$ coincide with $+$ and $\cdot$. This definition corresponds to the one in [10]. We will use $a, b, c \ldots$ and x, y, z for arbitrary S-elements and $p, q, r, \ldots$ for tests. By $\neg$ we denote complementation in $\mathsf{test}(S)$; implication $p \to q = \neg p + q$ obeys its standard laws.

A *domain semiring* is a pair $(S, {}^\ulcorner)$, where S is a test semiring and the *domain* operation ${}^\ulcorner : S \to \mathsf{test}(S)$ satisfies

$$a \leq {}^\ulcorner a \cdot a \quad \text{(d1)}, \qquad {}^\ulcorner(p \cdot a) \leq p \quad \text{(d2)}.$$

The relevant consequences are shown in [1]. In particular, domain is universally disjunctive and hence ${}^\ulcorner$ is strict, i.e., ${}^\ulcorner 0 = 0$. Furthermore we can expand (d1) to the equation $a = {}^\ulcorner a \cdot a$ (d1'). A corresponding codomain operation ${}^\urcorner : S \to \mathsf{test}(S)$ can defined analogously. S is called a *bidomain semiring* if there are domain and codomain operations. In bidomain semirings we have the following separability property:

$$a^\urcorner \cdot {}^\ulcorner b \leq 0 \Leftrightarrow a^\urcorner \cdot b \leq 0 \Leftrightarrow a \cdot {}^\ulcorner b \leq 0 \,. \qquad \text{(sep)}$$

In [8] we showed that the structure $\text{INT} = (\mathcal{P}(\mathbb{I}), \cup, ;, \emptyset, \mathbb{1})$ is an idempotent semiring, where $\mathbb{I} \overset{\text{def}}{=} \{[b, e] : b \leq e, b, e \in \mathbb{T}\text{ime}\}$ is the set of all intervals, $; : \mathcal{P}(\mathbb{I}) \times \mathcal{P}(\mathbb{I}) \to \mathcal{P}(\mathbb{I})$ defines the elementwise interval composition and $\mathbb{1} \overset{\text{def}}{=} \{[b, b] : b \in \mathbb{T}\text{ime}\}$ is the neutral element with respect to multiplication. The definition of interval composition says that $[a, b] ; [c, d]$ is defined if and only if $b = c$, i.e., iff the interval $[c, d]$ is part of the "right neighbourhood" of $[a, b]$, or, symmetrically, iff $[a, b]$ is part of the "left neighbourhood" of $[c, d]$. Here the domain (codomain) characterises the starting points (end points) of intervals, i.e., for $x \in \mathcal{P}(\mathbb{I})$

$$\ulcorner x = \{[b, b] : [b, e] \in x\} \qquad \text{and} \qquad x\urcorner = \{[e, e] : [b, e] \in x\} .$$

These operators imply the following view of the neighbourhood modalities.

$$\begin{aligned} \diamond_r \phi \text{ holds on } \{[b, e]\} &\Leftrightarrow \exists\ [u_1, u_2] \in \mathbb{I}_\phi : [b, e] \,;\, [u_1, u_2] \text{ is defined} \\ &\Leftrightarrow \{[b, e]\}\urcorner \leq \ulcorner(\mathbb{I}_\phi), \end{aligned}$$

In general we have for $\diamond_l \phi$ and $\diamond_r \phi$ the following equivalences:

$$\begin{aligned} \diamond_l \phi \text{ holds on } x \in \mathcal{P}(\mathbb{I}) \text{ iff } \ulcorner x \leq \mathbb{I}_\phi\urcorner \ , \\ \diamond_r \phi \text{ holds on } x \in \mathcal{P}(\mathbb{I}) \text{ iff } x\urcorner \leq \ulcorner\, \mathbb{I}_\phi \ . \end{aligned}$$

As a first result we show that at least one of the eight axioms, which are postulated in [14] can be dropped and is in fact a theorem in bidomain semirings. More simplifications on calculations are given in Section 4 after introducing a general form of neighbourhoods.

Lemma 3.1 $\diamond(\phi \vee \psi) \Leftrightarrow \diamond\phi \vee \diamond\psi$, *where* $\diamond$ *is* $\diamond_r$ *or* $\diamond_l$. *Hence Axiom 4 of [14] is a conclusion.*

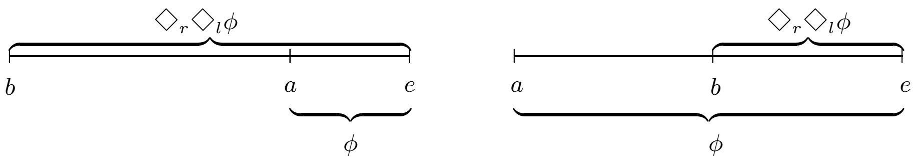
Now we will discuss the box operators $\Box_l \phi \overset{\text{def}}{=} \sim\diamond_l\sim\phi$ and $\Box_r \overset{\text{def}}{=} \sim\diamond_l\sim\phi$ of Zhou and Hansen in detachment and bidomain semirings, respectively. Here, $\sim$ is the negation of truth values, i.e., $\sim(\mathsf{true}) = \mathsf{false}$ and $\sim(\mathsf{false}) = \mathsf{true}$. In [13, 14] it is denoted as usual by $\neg$. But this symbol clashes with the negation symbol of tests. The meaning of $\Box_l \phi$ ($\Box_r \phi$) is:

$$\Box_l \phi (\Box_r \phi) \text{ holds on } [b, e] \Leftrightarrow \phi \text{ holds on all neighbours left (right) of } [b, e] \ .$$

In bidomain semirings we get a generalised form by:

$$\begin{aligned} \Box_l \phi \text{ holds on } x \in \mathcal{P}(\mathbb{I}) \text{ iff } (\overline{\mathbb{I}_\phi})\urcorner \,;\, \ulcorner x \leq 0 \quad \text{ and} \\ \Box_r \phi \text{ holds on } x \in \mathcal{P}(\mathbb{I}) \text{ iff } x\urcorner \,;\, \ulcorner(\overline{\mathbb{I}_\phi}) \leq 0 \ . \end{aligned}$$

In [14] the authors introduce the composed neighbourhood modalities $\diamond_r \diamond_l \phi$ and $\diamond_l \diamond_r \phi$ and called them *converses*. But these are very unhandy in calculations and we show that they are again diamonds closely related to $\diamond_l$ and $\diamond_r$. We want to illustrate the meaning of $\diamond_r \diamond_l \phi$. Here, we have that either $[a, e]$ is a postfix of $[b, e]$ or, if $a \leq b$, $[b, e]$ is a postfix of $[a, e]$:



where $a = e - \delta$.

Now we have a look at $\Diamond_r\Diamond_l\phi$ using domain and codomain.

$$\begin{aligned}
\Diamond_r\Diamond_l\phi \text{ holds on } x &\Leftrightarrow x^{\urcorner} \leq {}^{\ulcorner}(\mathbb{I}_{\Diamond_l\phi}) \\
&\Leftrightarrow x^{\urcorner} \leq {}^{\ulcorner}\{[b,e] \,:\, {}^{\ulcorner}\{[b,e]\} \leq \mathbb{I}_\phi{}^{\urcorner}\} \\
&\Leftrightarrow x^{\urcorner} \leq \mathbb{I}_\phi{}^{\urcorner}\ , \\
\Diamond_l\Diamond_r\phi \text{ holds on } x &\Leftrightarrow {}^{\ulcorner}x \leq {}^{\ulcorner}\mathbb{I}_\phi\ .
\end{aligned}$$

We see that $\Diamond_r\Diamond_l\phi$ and $\Diamond_l\Diamond_r\phi$ can be as easily expressed as the single diamonds introduced above. The four neighbourhood operators $(\Diamond_l, \Diamond_r, \Diamond_l\Diamond_r, \Diamond_r\Diamond_l)$ represent all combinations for comparing domain and codomain and therefore motivate the generalised definition in the next section.

4 Generalised Neighbourhoods and some Properties

Starting with the definitions of neighbourhoods given in Section 3 and motivated by NL we give general definitions, which work on bidomain semirings.

Definition 4.1 Let S be a bidomain semiring and $x, y \in S$. Then
(i) x is a *left neighbour* of y (or $x \leq \Diamond\mkern-11mu{\scriptstyle\eta}\mkern3mu_l y$ for short) iff $x^{\urcorner} \leq {}^{\ulcorner}y$,
(ii) x is a *right neighbour* of y (or $x \leq \Diamond\mkern-11mu{\scriptstyle\eta}\mkern3mu_r y$ for short) iff ${}^{\ulcorner}x \leq y^{\urcorner}$,
(iii) x is a *left boundary* of y (or $x \leq \Diamond\mkern-11mu{\scriptstyle\beta}\mkern3mu_l y$ for short) iff ${}^{\ulcorner}x \leq {}^{\ulcorner}y$,
(iv) x is a *right boundary* of y (or $x \leq \Diamond\mkern-11mu{\scriptstyle\beta}\mkern3mu_r y$ for short) iff $x^{\urcorner} \leq y^{\urcorner}$.

We will see below that the notation using $\leq$ is justified. Now we have a closer look at the definition and its interpretation in INT. For example (i) describes the situation, where for each element $[a,b]$ of x there exists at least one interval in y with starting point b. Hence $\Diamond_r\phi$ holds on x if and only if x is a left neighbour of $\mathbb{I}_\phi$ ($x \leq \Diamond\mkern-11mu{\scriptstyle\eta}\mkern3mu_l\mathbb{I}_\phi$). The change in direction (left, right) follows from the point of view. $\Diamond_r\phi$ starts with an interval of x and has a look at elements of $\mathbb{I}_\phi$ at its right, whereas our definitions start at $\mathbb{I}_\phi$. Starting at our definitions of neighbours and borders we calculate an explicite form of these operations.

Lemma 4.2 *Neighbours and boundaries can be expressed explicitly by*

$$\begin{aligned}
\Diamond\mkern-11mu{\scriptstyle\eta}\mkern3mu_l y &= \top \cdot {}^{\ulcorner}y\ , & \Diamond\mkern-11mu{\scriptstyle\eta}\mkern3mu_r y &= y^{\urcorner} \cdot \top\ , \\
\Diamond\mkern-11mu{\scriptstyle\beta}\mkern3mu_l y &= {}^{\ulcorner}y \cdot \top\ , & \Diamond\mkern-11mu{\scriptstyle\beta}\mkern3mu_r y &= \top \cdot y^{\urcorner}\ .
\end{aligned}$$

If there is a complementation function $\overline{}$ on S, which satisfies $\overline{\overline{a}} = a$, $\overline{a} + a = \top$ and $a \leq b \Leftrightarrow \overline{b} \leq \overline{a}$, we define perfect neighbours and perfect boundaries.

Definition 4.3 Let S be a complement bidomain semiring and $x, y \in S$.
(i) x is a *perfect left neighbour* of y (or $x \leq \boxed{\scriptstyle\eta}_l y$ for short) iff $x^{\urcorner} \cdot {}^{\ulcorner}\overline{y} \leq 0$,
(ii) x is a *perfect right neighbour* of y (or $x \leq \boxed{\scriptstyle\eta}_r y$ for short) iff $\overline{y}^{\urcorner} \cdot {}^{\ulcorner}x \leq 0$,
(iii) x is a *perfect left boundary* of y (or $x \leq \boxed{\scriptstyle\beta}_l y$ for short) iff ${}^{\ulcorner}x \cdot {}^{\ulcorner}\overline{y} \leq 0$,
(iv) x is a *perfect right boundary* of y (or $x \leq \boxed{\scriptstyle\beta}_r y$ for short) iff $x^{\urcorner} \cdot \overline{y}^{\urcorner} \leq 0$.

By (iii) and (iv) we have an additional extension of NL. These two definitions define "box-operators" for the converses of neighbourhood modalities, which are not defined in the semantics of NL given in [13]. To justify the definitions above we have

Lemma 4.4 *Each perfect neighbour (boundary) is a neighbour (boundary).*

We can characterise the box operations, like neighbours/boundaries, in an explicit form.

Lemma 4.5 *Perfect neighbours and perfect boundaries have the following explicit forms:*

$$\begin{array}{ll} \boxed{\eta}_l y \;=\; \top \cdot \neg^{\ulcorner}(\overline{y}) \;, & \boxed{\eta}_r y \;=\; \neg(\overline{y})^{\urcorner} \cdot \top \;, \\ \boxed{\beta}_l y \;=\; \neg^{\ulcorner}(\overline{y}) \cdot \top \;, & \boxed{\beta}_r y \;=\; \top \cdot \neg(\overline{y})^{\urcorner} \;. \end{array}$$

To reduce calculations we introduce $\Diamond$ and $\boxdot$ as parameters, which can be instantiated by either $\langle\eta\rangle_l$, $\langle\eta\rangle_r$, $\langle\beta\rangle_l$ or $\langle\beta\rangle_r$ and $\boxed{\eta}_l$, $\boxed{\eta}_r$, $\boxed{\beta}_l$ or $\boxed{\beta}_r$, respectively. If the "direction“ of $\Diamond$ or $\boxdot$ is important we use formulas like $\Diamond_l$ and $\boxdot_r$ where only one degree of freedom remains.
Boxes and diamonds are connected via the de Morgan duality

$$\boxdot y \;=\; \overline{\Diamond\overline{y}},$$

hence they form proper modal operations. Additionally, it follows that diamonds and boxes are lower and upper adjoints of Galois connections:

$$\Diamond_l x \le y \;\Leftrightarrow\; x \le \boxdot_r y \;, \qquad \Diamond_r x \le y \;\Leftrightarrow\; x \le \boxdot_l y \;.$$

By the Galois connections and de Morgan dualities we get many properties of (perfect) neighbours and (perfect) boundaries for free. For example we have, with $x \sqcap y \;=\; \overline{\overline{x} + \overline{y}}$,

Corollary 4.6

(i) $\Diamond$ *and* $\boxdot$ *are isotone.*
(ii) $\Diamond$ *is distributive and* $\boxdot$ *is conjunctive,*
i.e., $\Diamond(x+y) \;=\; \Diamond x + \Diamond y$ *and* $\boxdot(x \;\sqcap\; y) \;=\; \boxdot x \;\sqcap\; \boxdot y$.
(iii) We also have the cancellative laws
$\Diamond_l \boxdot_r x \le x \le \boxdot_r \Diamond_l x$ *and* $\Diamond_r \boxdot_l x \le x \le \boxdot_l \Diamond_r x$.

In sum, all theorems given in [13, 11, 14] hold in the generalisation, too. Most of them are already proved by the Galois connection and the Corollary above.

Lemma 4.7

(i) $\langle\eta\rangle_l \langle\eta\rangle_r y \;=\; \langle\eta\rangle_l y$ *and* $\langle\eta\rangle_r \langle\eta\rangle_l y \;=\; \langle\eta\rangle_r y$.
(ii) $\langle\eta\rangle_l \langle\eta\rangle_r y \le \boxed{\eta}_l \langle\eta\rangle_r y$ *and* $\langle\eta\rangle_r \langle\eta\rangle_l y \le \boxed{\eta}_r \langle\eta\rangle_l y$.
(iii) $\boxed{\eta}_l \boxed{\eta}_r y \;=\; \boxed{\eta}_r y$ *and* $\boxed{\eta}_r \boxed{\eta}_l y \;=\; \boxed{\eta}_l y$

Lemma 4.7.(ii) is the same as Axiom 6 of [14], which is now a theorem. There are many more simplifications and extensions for NL which we do not discuss here.

Interpretation in other models

We generalised NL to arbitrary bidomain semirings. Thus we are able to adopt the theory to other areas. Bidomain semirings having applications in computer science are for example

- REL, the algebra of binary relations over a set under relational composition,
- LAN, the algebra of formal languages under language concatenation, and
- PAT, the algebra of sets of graph paths under path fusion.

More semirings and applications can be found e.g. in [1, 6]. In all these semirings we can interpret (perfect) neighbours and (perfect) boundaries. In PAT for example, ${\Diamond\!\!\!\!\!{\scriptstyle\eta}}\;_r T$ is the set of all paths which can be reached from the paths in T. In contrast to this, ${\Box\!\!\!\!\!{\scriptstyle\eta}}\;_r T$ is the set of all paths which can only be reached from T.

In a discrete semiring S, i.e., $\mathsf{test}(S) = \{0, 1\}$, like LAN, all diamonds (${\Diamond\!\!\!\!\!{\scriptstyle\eta}}\;_l$, ${\Diamond\!\!\!\!\!{\scriptstyle\eta}}\;_r$, ${\Diamond\!\!\!\!\!{\scriptstyle\beta}}\;_l$, ${\Diamond\!\!\!\!\!{\scriptstyle\beta}}\;_r$) are the same and all boxes collapses, too. We have

$$\Diamond\!\!\!\!\!\cdot\;\; L = \begin{cases} 0 \text{ if } L = \emptyset \\ \top \text{ otherwise,} \end{cases} \qquad\qquad \boxdot L = \begin{cases} \top \text{ if } L = \top \\ 0 \text{ otherwise.} \end{cases}$$

Based on INT we presented an embedding of the Duration Calculus in idempotent semirings in [8]. Here we can adopt the theory, too. In [7, 9] we introduced an algebra of processes, where processes are sets of trajectories. These models are a first step towards the description of hybrid systems in an algebraic manner. The right neighbour ${\Diamond\!\!\!\!\!{\scriptstyle\eta}}\;_r$ characterises properties of trajectories which will be reached in the future. More informations concerning more details about the interpretations of neighbours/boundaries as well as interpretations in other models can be found in [7].

5 Conclusion and Outlook

In this paper we started with the Neighbourhood Logic developed by Zhou and Hansen. We showed that we can embed NL into the theory of semirings. With the help of the embedding we showed that at least two axioms can be dropped in the definition of NL and that neighbours can be expressed in a much more general framework. Therefore we presented neighbours and boundaries in bidomain semirings and presented important Galois connections. At the end we gave a short discussion for further applications of the generalised version of NL.

Möller developed the theory of Lazy semirings and we presented an algebra for hybrid systems in [9]. Thus we want to adapt and, if necessary, modify the neighbours and boundaries to Lazy semirings. Then we have a further application for NL in a theory where we can express unlimited processes.

Acknowledgements I am very grateful to Bernhard Möller for his valuable comments and remarks, especially concerning the explicit formulations of neighbours. Many useful comments were also provided by Kim Solin.

References

1. J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Transaction on Computational Logic*, 2004.
2. B. Dutertre. Complete proof systems for first-order interval temporal logic. In IEEE Press, editor, *Tenth Annual IEEE Symb. on Logic in Computer Science*, pages 36–43, 1995.
3. J.S. Golan. The Theory of Semirings with Applications in Mathematics and Theoretical Computer Science. *Longman*, 1992. ISBN 0-582-07855-5.
4. J.Y. Halpern, B. Moszkowski and Z. Manna. A Hardware Semantics based on Temporal Intervals. *In: Diaz J. (editors) ICALP'83, Lecture Notes in Computer Science 154.*:278–291. Springer, Berlin, 1983.
5. J.Y. Halpern and Y. Shoham. A Propositional Modal Logic of Time Intervals. Proceedings of the First IEEE Symposium on Logic in Computer Science. IEEE Press, Piscataway, NJ, 279–292.
6. U. Hebisch, and H.J. Weinert. Semirings - Algebraic Theory and Applications in Computer Science. *World Scientific, Singapur*, 1998. ISBN 981-02-3601-8.
7. P. Höfner. *Semiring Neighbours.* Technical Report 2005-19, Universität Augsburg, 2005.
8. P. Höfner. *From Sequential Algebra to Kleene Algebra: Interval Modalities and Duration Calculus.* Technical Report 2005-05, Universität Augsburg, 2005.
9. P. Höfner and B. Möller. Towards an Algebra of Hybrid Systems. *Proceedings of the 8th International Conference on Relational Methods in Computer Science (RelMiCS 8)*, February 2005.
10. D. Kozen. Kleene algebra with tests. *Trans. Prog. Languages and Systems*, 19(3): 427–443, 1997.
11. S. Roy and C.C. Zhou. Notes on Neighbourhood Logic. Technical Report 97, The United Nations University UNU/IIST, February 1997.
12. Y. Venema. A Modal Logic for Chopping Intervals. *J. of Logic and Computation* 1(4):453–476, 1990.
13. C.C. Zhou and M.R. Hansen. *Duration Calculus – A Formal Approach to Real-Time Systems.* Springer, Monographs in Theoretical Computer Science, 1996.
14. C.C. Zhou and M.R. Hansen. *An Adequate First Order Interval Logic. Lecture Notes in Computer Science*, 1536:584–608, 1998.

Linking π-calculus and B-Method

Damien Karkinsky

School of Electronics and Physical Sciences,
University of Surrey, Guildford, Surrey GU2 7XH
d.karkinsky@surrey.ac.uk

Abstract. The approach we consider in this paper is an integration of the π-calculus and the B-Method. In order to be able to reason about specifications based on both these notations we need common semantics. We illustrate an approach which enables the interpretation of a B machine as a π labelled transition system so that we can consider parallel combinations of π-calculus processes and B machines. The benefit of this work is the extension of B machines with instantiation and π-calculus dynamic reconfiguration capabilities.

1 Introduction

We are interested in combinations of formal methods which consider the state and dynamic requirements of a system. We recognise that many such combinations already exist, including, *CSP* || *B* [7] and *Circus* [2], but we are concerned with a state description, being accessed and updated by control components with dynamically reconfigurable interconnections. Our work is motivated by what we see in object-oriented modelling where object instantiation and dynamically reconfigurable interconnection are essential paradigms. For example, an object instance can be created with a unique reference. This reference can be used by other objects to communicate with the instance, but we must be careful to control the way the instance is updated and read by objects. Our aim is to provide a formal framework which supports this kind of interaction so that the integrity of each active object is preserved, and so that we can reason about the overall behaviour of the system.

The approach we consider in this paper is a combination of the π-calculus [6] and the B-Method [1]. In order to be able to reason about specifications based on both these notations we need common semantics. We illustrate an approach which enables the interpretation of a B machine as a π labelled transition system so that it can be integrated into parallel combinations with π-calculus processes. As a result, this work naturally extends B machines with instantiation and π-calculus dynamic reconfiguration capabilities.

This paper is organised as follows. A short overview of each method is presented in the following Section. Section 3.1 presents a standard LTS interpretation for machines, which we make use of in the π-semantics for B machines in Section 4.

2 Background

The π-calculus syntax is based on an infinite set of names $\mathcal{N}$, which can represent both values and channels for communication. This allows the treatment of channels as values in communications over other channels.
Structurally the π-calculus is very minimal, containing constructs for; an inactive process $\mathbf{0}$, concurrent execution $P_1 \ | \ P_2$ (where each P is a π processes), external choice $P_1 \ + \ P_2$, channel scoping $(\nu \ v)(P)$ where the scope of channel v is P, infinite replication $!P$, and action prefix $\pi.P$. Action prefixes can be; input $a(w)$, output $\overline{a}w$, or internal actions τ, each of which can be guarded using matching, which is not considered in this paper. The names a, w are members of $\mathcal{N}$.
The syntax is given an operational semantics which is defined in terms of three binary relations on π processes a structural congruence ($\equiv_\pi$), a reduction relation ($\longrightarrow$), and a labelled transition relation ($\xrightarrow{\alpha}$). A full definition of these relations can be found in [6]. They define; communication between concurrent agents as binary and synchronous, and support processes generating *fresh* channels at runtime and extruding them outside their scope through output actions.
In our paper, we use two extensions to the core π-calculus syntax presented above. First, is the variant construct $case \ o \ of \ [l_1 \triangleright P_1; \quad \ldots \ l_n \triangleright P_n]$ which evolves to P_i for $1 \leq i \leq n$ whenever o matches the label l_i. This construct was originally introduced for the study of π with respect to object oriented languages (see [6]p252). Second, is a form of indexing called pseudo-application; the expression $P = (x).Q$ denotes an abstraction of the definition of agent Q where x might be a free name. Given a name v, $Q\{^v/_x\}$ is an instance of P where v is substituted for every free occurrence of x therefore, we can denote $Q\{^v/_x\}$ with $P\langle v \rangle$.
The B-method is a first order predicate calculus language with set theory. The main specification construct is a *MACHINE* [1] consisting of an initialisation and a set of operations(in this paper restricted to ones without I/O and guards). An example machine *LightSwitch*, is defined in [4] with initial state $nn = 0$ and two operations: $on \ \widehat{=} \ PRE \ \ nn = 0 \ \ THEN \ \ nn := 1 \ \ END$, and an inverse of on, called off.

3 Preliminary labelled transition system

We define LTS_M to be a labelled transition system for a given B machine M, as follows.

Definition 3.1.

$$LTS_M = (ST_M^{\perp}, \ \ OPERATIONS, \ \ INIT_M, \ \ \longrightarrow_M)$$

$ST_M^{\perp}$ is the state space of M together with a divergent state $\perp$. $OPERATIONS$ is a set of labels one for each operation of the machine. $INIT_M$ is a set of initial states from ST_M. $\longrightarrow_M$ is a set of triples from $(ST_M^{\perp} \times OPERATIONS \times ST_M^{\perp})$

containing the state transitions of the operations.
A distinguishing feature of LTS_M is that ST_M is a set of *valuations* satisfying the machine invariant. A valuation is a total function from the *VARIABLES* of the machine to the value domain for the B language which we call D_B. The set of valuations $INIT_M$ and the transition relation $\longrightarrow_M$ are defined in [5], using predicates $abt(S)$ (aborting) and $prd(S)$ (pre & post) defined for machine statements S, in [1]. We can show that LTS_M is consistent with *failure* & *divergence* semantics for action systems [4] with guards set to *true*.

4 *Π* style labelled transition system for B-Machines

This section explains how LTS_M can be integrated with the π operational semantics. Firstly, the labels of a variant construct are matched with the *OPERATIONS* labels of LTS_M. For example, π variant labels representing the B operations *on* and *off* are $on\_\langle * \rangle$ and $off\_\langle * \rangle$ respectively and define $V_M = \{on\_\langle * \rangle,\ \ off\_\langle * \rangle\}$[1]. The $*$ means there is no I/O. Labels can be sent on given channels, such as z, from the π-specifications to an instance of *LightSwitch* machine e.g. $\overline{z}\, on\_\langle * \rangle$. The channel z can be viewed as a reference to a B object.
Secondly, we extend the state-space ST_M (without $\perp$) adding points of machine activity: *BEGIN* is the point when M is not initialised, *READY*- ready to execute an operation, and for each operation of M, $BODY_{op_i}$, is when op_i is being executed. If $IS = \{BEGIN,\ \ READY, BODY_{op_1},\ \ \ldots\ ,\ \ BODY_{op_n}\}$ then the extended state-space is given by the following definition:

Definition 4.1. $ST_\pi = IS \times ST_M$.

Since the state of the variables of M is not important at point *BEGIN* we denote $(BEGIN, val)$ for any val as $(BEGIN)$.
Thirdly, we define a system of π-processes $[\![s]\!]\langle z \rangle$ where $[\![s]\!]$ denotes a process abstraction parameterised by channel z and $s \in ST_\pi$. We define the behaviour of $[\![s]\!]\langle z \rangle$ using reduction and labelled transition rules in Def 4.2. This defines how a B object interacts with a π-environment.

Definition 4.2. *Given* LTS_M *and a channel* z*,*

$$\begin{aligned}
&1 : [\![(BEGIN)]\!]\langle z \rangle \ \ (\longrightarrow)^+ \ \ [\![(READY, val_1)]\!]\langle z \rangle \ \ \textit{iff}\ val_1 \in INIT_M \\
&2 : [\![(READY, val)]\!]\langle z \rangle \ \ \xrightarrow{z\ l} \ \ [\![(BODY_{op(l)}, val)]\!]\langle z \rangle \\
&3 : [\![(BODY_{op(l)}, val)]\!]\langle z \rangle \ \ (\longrightarrow)^+ \ \ [\![(READY, val_1)]\!]\langle z \rangle \\
&\qquad\qquad \textit{iff}\ (val, op(l), val_1) \in\longrightarrow_M \quad \textit{and}\ val_1 \neq \perp \\
&4 : [\![(BODY_{op(l)}, val)]\!]\langle z \rangle \ \ \longrightarrow \ \ \Omega\ \textit{if}\ (val, op(l), \perp) \in\longrightarrow_M
\end{aligned}$$

where $op(l) \in OPERATIONS$ *and* $l \in V_M$

[1] V_M is a variant type derived from the machine signature and used in the type-checking of π-agents. The type system we use is [6]p289.

$$\begin{tikzcd}
{[\![(BEGIN)]\!]\langle z\rangle} \arrow[d, "(\tau)^+"'] & \\
{[\![(READY,\{(nn,0)\})]\!]\langle z\rangle} \arrow[r, "z\ \ \mathit{off}\_\langle * \rangle"] \arrow[d, "z\ \ on\_\langle * \rangle"] & {[\![(BODY_{\mathit{off}},\{(nn,0)\})]\!]\langle z\rangle} \arrow[d, "\tau"] \\
{[\![(BODY_{on},\{(nn,0)\})]\!]\langle z\rangle} \arrow[d, "(\tau)^+"'] & \Omega \arrow[loop left, "\tau"] \\
{[\![(READY,\{(nn,1)\})]\!]\langle z\rangle} \arrow[r, "z\ \ on\_\langle * \rangle"] \arrow[d, "z\ \ \mathit{off}\_\langle * \rangle"] & {[\![(BODY_{on},\{(nn,1)\})]\!]\langle z\rangle} \arrow[u, "\tau"'] \\
{[\![(BODY_{\mathit{off}},\{(nn,1)\})]\!]\langle z\rangle} \arrow[uuu, bend left, "(\tau)^+"] &
\end{tikzcd}$$

PSfrag replacements

Fig. 1. π LTS of *LightSwitch*

Within the π-calculus every reduction matches with a τ action up to structural congruence see [6]p51. Thus, we can restate reduction rules in Def 4.2.$(1, 3, 4)$ as labelled transition rules where each reduction is replaced with a τ transition. Def 4.2.1 represents the initialisation of the machine. Def 4.2.2 represents receiving an operation call along channel z. It can be shown that $[\![(READY, val)]\!]\langle z\rangle$ has the same transitions as the following process abstraction if channel *getop* is instantiated with z.

$$(getop).getop(o).case\ \ o\ \ of\ \ [1 \leq i \leq n,\ \ op_i\_\langle * \rangle \triangleright [\![(BODY_{op_i}, val)]\!]\langle getop\rangle;\ ]$$

This justifies the input action $z\ \ l$ as an appropriate labelled action.
Def 4.2.3 is a correct execution of the corresponding operation. Def 4.2.4 represents calling an operation outside its pre-condition, where Ω is a constant agent that can perform an infinite number of τ actions. Notice that after initialisation one non-divergent transition of LTS_M is decomposed into an input action followed by a positive finite number of τ transitions in the π LTS[2].
Considering machine *LightSwitch*, using Def 4.2 we can generate the π-calculus LTS in Fig 4. We can also instantiate the abstraction $[\![(BEGIN)]\!]$ as a regular π-agent within π-specifications. However, we only wish to consider instances where the machine is executed concurrently with other agents. For example,

$$!(\nu\ \ z)(\overline{createM}\langle z\rangle.0\ \ |\ \ [\![(BEGIN)]\!]\langle z\rangle) \tag{1}$$

[2] This consideration, on one side, is imposed by the granularity of π-actions and becomes important in machines with I/O.

The agent above can be interpreted as an infinite collection of machine instances. It can be shown that each B instance has its own unique reference channel that becomes available after interaction on *createM*.

5 Discussion

The work [8], that relates the π-calculus and Object-Z, differs to our approach in several respects. In [8] the specifications are a mixture between a control model and a state model of a given system. Pi/Object-Z (PiOZ) class definitions are given in terms of simple Object-Z schemas and π-like process definitions. We use the standard π-calculus whereas, they change the syntax and interpretation. For example, channels are viewed as state variables, and action prefixing is replaced with sequential composition of processes. Secondly, their interpretation of an operation call is the execution of a process, whereas ours is a labelled action.
The motivation behind separating π-calculus and B descriptions cleanly came from the CSP$\|$B approach [7]. In CSP$\|$B a B machine is owned by a specific CSP controller and the specification cannot evolve dynamically so that another process can take over control of the B machine. The benefit of the approach presented here is that we have a clearer way of passing control over B objects from one π-calculus controller to another.
Although this paper considered machines without I/O the approach can be extended to accommodate it. The extension requires that output from the machine, is transmitted separately from operation calls and temporary channels can be established during each operation call.
We are currently developing a trace based/assertional verification method which can be used to show divergence freedom properties over the machine instances.

References

1. Abrial J-R. *The B-Book, Assigning Programs to Meanings* Cambridge University Press 1996.
2. Cavalcanti A., Sampaio A., and Woodcock J.: *Refinement of Actions in Circus*, In REFINE'02, FME Workshop, Copenhagen (2002).
3. Dijkstra E. W. *A Discipline of Programming* Prentice-Hall Inc, 1976.
4. Morgan C.C. *On wp and CSP* In W.H.J. Feijen, A.J.M. van Gasteren, D Gries and J.Misra, editors, Beauty is our business: a birthday salute to Edsger W. Dijkstra. Springer 1990.
5. Karkinsky D. *Using B-Method and π-calculus (in preparation)*, 2006
6. Sangiorgi D., Walker D. *The π-calculus, A Theory of Mobile Processes* Cambrige University Press 2001.
7. Schneider, S., Treharne, H.: *CSP Theorems for Communicating B Machines.* In Proceedings of IFM 2004, LNCS 2999, Springer-Verlag, University of Kent, 2004.
8. Taguchi K., Dong J. S., Ciobanu G. *Relating π-calculus to Object-Z* p97-106 ICECCS 2004.

Noninterference for sequential and multi-threaded languages

Thuy Duong Vu

Programming Research Group, University of Amsterdam
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
`tdvu@science.uva.nl`

Abstract. Several definitions of noninterference for sequential programming languages have been introduced. Unfortunately, these definitions appear confusing when parallelism is introduced in the languages. We propose a method to verify noninterference in both sequential and multi-threaded programming languages. Noninterference is defined in the setting of Basic Polarized Process Algebra (BPPA), a natural candidate for the specifications of sequential and multi-threaded program semantics [Bergstra, Loots in 2002], [Bergstra, Middelburg in 2004]. More precisely, a program is secure if its behavior cannot be interfered with its high-security (or implicit) actions. Our definitions of noninterference are simple, and satisfy certain desirable properties.

1 Introduction and related works

In 2002, Bergstra and Loots introduced *Basic Polarized Process Algebra* (BPPA) [4], an algebraic theory about sequential program behaviors. Later, Bergstra and Middelburg [3] proposed an extension of BPPA, called *thread algebra*, as a framework for the description and analysis of multi-threaded strategic interleaving. It has been outlined in [4, 3] how and why these algebras are natural candidates for the specifications of sequential and multi-threaded program semantics. This paper considers *noninterference* for sequential and multi-threaded languages, in which program behaviors are described in the setting of BPPA.

Noninterference [7] is a property of secure information flow [2, 6] that characterizes programs whose execution does not reveal information about secret data. More precisely, program variables are classified into two disjoint security levels "low-security" and "high-security". We say that such a program is *noninterferent* if observations of the initial and final values of low (or public) variables do not provide any information about the initial values of high (or secret) variables.

Although this problem has been studied for several decades, most of the previous approaches for sequential and multi-threaded languages have been syntactic in nature, often using the so-called *type system* technique to analyze program texts [14, 13, 12, 10, 5] (for an overview, see [11]). However, such type systems are imprecise and complicated. Furthermore, they reject many secure programs, even for simple programming languages.

Some recent approaches [9, 1] consider security in various logical forms, leading to a characterization using Hoare triples. These approaches give a more precise characterization of security than the previous ones. But, they still have limitations. For instance, for programs with iteration or recursion, determining security would require complex computations, which makes this checking unattractive in practice. Moreover, both classes of approaches have problems when parallelism is introduced in the languages. Therefore, they are not suitable candidates when dealing with multi-threading.

In this paper, we first formulate the notion of *noninterference* given by Goguen and Meseguer [7] for polarized processes in BPPA, using input-output transformations. This definition is precise. However, the checking of this noninterference would require complex computations. To simplify this checking, we propose several definitions of noninterference based on program behaviors. Informally speaking, a program is secure if its behaviors cannot be interfered with *high* (or *implicit*) actions. We extend BPPA to *Secure Basic Polarized Process Algebra* (SBPPA) with an internal action t, the *hiding* and *abstraction* operators. We define *branching bisimulation* for polarized processes which classifies processes that behave the same, from the view of non-internal actions. Using this bisimulation equivalence, we define *behavioral noninterference* (BNI) for polarized processes in BPPA. We prove the soundness theorem: If a process is behaviorally noninterferent then it is noninterferent. In the setting of thread algebra, we present a strategic interleaving operator for multi-threaded systems, called *cyclic interleaving with persistence* to turn a thread vector of arbitrary length into a single thread (assuming that a thread is a process). The advantage of this operator, in comparison with the operators defined in [3], is that branching bisimulation is a *congruence* with respect to this operator. We prove that the definition of noninterference based on input-output transformations satisfies *separability* [8]. We also consider *compositionality* and *decompositionality* of polarized processes under cyclic interleaving with persistence. It turns out that our definition of behavioral noninterference is closed under this operator. Furthermore, it satisfies a decompositional property for finite processes.

2 Secure Basic Polarized Process Algebra (SBPPA)

2.1 Primitives of BPPA and input-output transformations

Let Σ be the set of *basic actions*. Each basic action is supposed to return a boolean value after its execution. *Basic Polarized Process Algebra* (BPPA) is defined with the following meaning:

- *Termination*, denoted by S, yields successful terminating behavior.
- *Deadlock*, denoted by D, represents inaction behavior.
- *Postconditional composition*: The process $P \unlhd a \unrhd Q$, where $a \in \Sigma$, first performs a and then proceeds with P if *true* was returned and with Q otherwise.

– *Action prefix*: For each $a \in \Sigma$ and process P,

$$a \circ P = P \trianglelefteq a \trianglerighteq P.$$

Let $BPPA_\Sigma$ be the set of *finite* processes which are made from S and D by means of a finite number of applications of postconditional compositions. To define *infinite* processes in *BPPA* we require a sequence of its finite approximations. For every $n \in \mathbb{N}$, the *approximation operator* $\pi_n : \mathrm{BPPA}_\Sigma \to \mathrm{BPPA}_\Sigma$ is defined inductively by

$$\begin{aligned}
\pi_0(P) &= D,\\
\pi_{n+1}(S) &= S,\\
\pi_{n+1}(D) &= D,\\
\pi_{n+1}(P \trianglelefteq a \trianglerighteq Q) &= \pi_n(P) \trianglelefteq a \trianglerighteq \pi_n(Q),
\end{aligned}$$

A *projective sequence* is a sequence $(P_n)_{n\in\mathbb{N}}$ such that for each $n \in \mathbb{N}$,

$$\pi_n(P_{n+1}) = P_n.$$

We now give a brief introduction about *input-output transformations* based on polarized processes. Let *Var* be the set of program variables. Suppose that upon the execution of a program, the value of a program variable is stored in a *memory*. A *state space* V is a deterministic space whose components are memories. For a state space V and an action a, the operation $\mathit{effect}_a(-) : V \to V$ that changes the memories due to the execution of a is so-called *effect* operation, while the operation $y_a(-) : V \to \{\mathit{true}, \mathit{false}\}$ that determines the boolean reply $y_a(v)$ produced when a is performed in a state of V, is called *yield* operation. Let D represent a failure value that can't be computed. A function $P \bullet (-) : V \to V \cup \{D\}$ which represents what P computes on input v in V is defined inductively as follows.

1. $D \bullet v = D$,
2. $S \bullet v = v$,
3. $(a \circ P) \bullet v = P \bullet \mathit{effect}_a(v)$,
4. $P \trianglelefteq a \trianglerighteq Q \bullet v = (a \circ P) \bullet v$ if $y_a(v)$, otherwise $(a \circ Q) \bullet v$.

We take $P \bullet v = D$ in order to express that the computation produces no result. In other words, $P \bullet v = D$ precisely if for all $n \in \mathbb{N}$, $\pi_n(P) \bullet v = D$. For a space $V = [V_1, V_2]$ with V_1, V_2 subspaces of V, and a value $v = [v_1, v_2] \in V$, $v|_{V_i}$ denotes the value v_i when we project v into V_i.

2.2 SBPPA

Secure Basic Polarized Process Algebra (SBPPA) is an extension of BPPA with an *internal action* t, the *hiding* and *abstraction* operators. Furthermore, the set Σ of basic actions is restricted to the set Σ_s of *secure* actions.

We suppose that program variables are classified into two security classes Var_L (*low* or *public*) and Var_H (*high* or *secret*) ($Var_L \cap Var_H = \emptyset$). Moreover, a deterministic state space V consists of two subspaces: low (or public) V_L and high (or secret) V_H. Each component of V_L stores the value of a low variable, and each component of V_H stores the value of a high variable in a program. Let $\mathit{var}(V)$ and $\mathit{var}(a)$ denote the set of program variables whose values are stored in V and the set of program variables occurring in a, respectively. We assume that if $\mathit{var}(a) \nsubseteq \mathit{var}(V)$ then $\mathit{effect}_a(v) = v$ and $y_a(v) = \mathit{false}$ for all $v \in V$. Furthermore, if $V = [V_1, \ldots, V_n]$ and $\mathit{var}(a) \subseteq \mathit{var}(V_i)$ for some i then $\mathit{effect}_a([v_1, \ldots, v_n]) = [v_1, \ldots, \mathit{effect}_a(v_i), \ldots, v_n]$ and $y_a([v_1, \ldots, v_n]) = y_a(v_i)$. A basic action is *secure* if $\mathit{effect}_a(v)|_{V_L} = \mathit{effect}_a(v|_{V_L})$ for all $v \in V$. A secure action is *implicit* if it has effect or yield on V_H, i.e. $\mathit{effect}_a(v)|_{V_H} \neq v|_{V_H}$ or $y_a(v) \neq y_a(v|_{V_L})$. Let Σ_s and H be the set of secure and implicit actions, respectively.

Let P be a polarized process. If $P = P_1 \trianglelefteq a \trianglerighteq P_2$ then P_1, P_2 are *successors* of P. A process Q is a *descendant* of P in a set $\mathbb{T}$ of processes if there is a sequence $P = P_0, P_1, \ldots, P_n = Q$ with $n \geq 0$ such that for all $0 \leq i \leq n$, $P_i \in \mathbb{T}$, and for all $0 \leq i < n$, P_{i+1} is a successor of P_i. A polarized process P is *divergent* if it has a descendant Q that diverges immediately, i.e., there is an infinite sequence $Q_0, Q_1, \ldots$ with $Q_0 = Q$ such that for all $n \in \mathbb{N}$: $Q_n = Q_{n+1} \trianglelefteq t \trianglerighteq Q'$ or $Q_n = Q' \trianglelefteq t \trianglerighteq Q_{n+1}$ for some process Q'. A *t-tree* of P is a set $\mathit{t\text{-}tree}(P)$ of processes containing P in which a process cannot be a descendant of itself. Moreover, if Q_1 is a process in $\mathit{t\text{-}tree}(P)$ and $Q_1 \neq P$, then there exist processes Q and Q_2 in $\mathit{t\text{-}tree}(P)$ such that $Q = Q_1 \trianglelefteq t \trianglerighteq Q_2$ or $Q = Q_2 \trianglelefteq t \trianglerighteq Q_1$. A process Q is a *leaf* of $\mathit{t\text{-}tree}(P)$ if $Q = S$ or $Q = D$ or $Q = Q_1 \trianglelefteq a \trianglerighteq Q_2$ with $Q_1, Q_2 \notin \mathit{t\text{-}tree}(P)$. A *branching bisimulation* relation $\mathcal{B}$ is a symmetric binary relation on processes in *SBPPA* satisfying:

1. If $(P, Q) \in \mathcal{B}$ and $P = S$, then there exists a finite $\mathit{t\text{-}tree}(Q)$ such that for every leaf Q' of $\mathit{t\text{-}tree}(Q)$: $(P, Q') \in \mathcal{B}$ and $Q' = S$.
2. If $(P, Q) \in \mathcal{B}$ and $P = P_1 \trianglelefteq a \trianglerighteq P_2$ then either:
   (a) $a = t$, and $(P_1, Q), (P_2, Q) \in \mathcal{B}$, or:
   (b) there exists a finite $\mathit{t\text{-}tree}(Q)$ such that for every leaf Q' of $\mathit{t\text{-}tree}(Q)$: $(P, Q') \in \mathcal{B}$, and $Q' = Q'_1 \trianglelefteq a \trianglerighteq Q'_2$ with $(P_1, Q'_1), (P_2, Q'_2) \in \mathcal{B}$.

Two processes P and Q are *branching bisimilar*, denoted by $(P \leftrightarroweq_b Q)$, if there is a branching bisimulation relation $\mathcal{B}$ such that $(P, Q) \in \mathcal{B}$.

We now introduce the notions of the hiding and abstraction operators. The *hiding operator* t_H : *BPPA* $\rightarrow$ *SBPPA* renames all implicit actions to the basic internal action t:

$$\begin{aligned}
t_H(S) &= S,\\
t_H(D) &= D,\\
t_H(P \trianglelefteq a \trianglerighteq Q) &= t_H(P) \trianglelefteq t \trianglerighteq t_H(Q) \text{ for } a \in H,\\
t_H(P \trianglelefteq a \trianglerighteq Q) &= t_H(P) \trianglelefteq a \trianglerighteq t_H(Q) \text{ otherwise.}
\end{aligned}$$

While the *abstraction* operator τ_H : *SBPPA* $\rightarrow$ *BPPA* removes these implicit actions from a process:

$$\begin{aligned}
\tau_H(S) &= S,\\
\tau_H(D) &= D,\\
\tau_H(P \trianglelefteq t \trianglerighteq Q) &= \tau_H(P) && \textit{for } a \in H,\\
\tau_H(P \trianglelefteq a \trianglerighteq Q) &= \tau_H(P) \trianglelefteq a \trianglerighteq \tau_H(Q) && \textit{for } a \notin H.
\end{aligned}$$

3 Thread algebra

Thread algebra is an extension of BPPA, which is designed for strategic interleaving of parallel processes. A *thread* is considered as a polarized process. A *thread vector* is a sequence of threads. *Strategic interleaving operators* turns a thread vector of arbitrary length into a single thread. This single thread obtained via a strategic interleaving operator is also called a *multi-thread*. Thread algebra is meant to specify the collection of strategic interleaving operators, capturing some essential aspects of multi-threading. In this section we introduce the *cyclic interleaving with persistence operator*, denoted by $\|_{csi}^{\Phi}$ for some $\Phi \subseteq \Sigma$, which is a slight modification of the *cyclic interleaving operator* ($\|_{csi}$) given in [3]. This operator invokes rotation of a thread vector only in the case that the current action is not in the set $\Phi \subseteq \Sigma$. Let $\langle\rangle$ denote for the empty sequence, $\langle x\rangle$ stand for a sequence of length one, and $\alpha \frown \beta$ for the concatenation of two sequences. We assume that the following identity holds: $\alpha \frown \langle\rangle = \langle\rangle \frown \alpha = \alpha$. The axioms for $\|_{csi}^{\Phi}$ are given as follows:

$$\begin{array}{ll}
\|_{csi}^{\Phi}(\langle\rangle) & = S \\
\|_{csi}^{\Phi}(\langle S\rangle \frown \alpha) & = \|_{csi}^{\Phi}(\alpha) \\
\|_{csi}^{\Phi}(\langle D\rangle \frown \alpha) & = S_D(\|_{csi}^{\Phi}(\alpha)) \\
\|_{csi}^{\Phi}(\langle x \unlhd a \unrhd y\rangle \frown \alpha) & = \|_{csi}^{\Phi}(\langle x\rangle \frown \alpha) \unlhd a \unrhd \|_{csi}^{\Phi}(\langle y\rangle \frown \alpha) \ \textit{for } a \in \Phi \\
\|_{csi}(\langle x \unlhd a \unrhd y\rangle \frown \alpha) & = \|_{csi}(\alpha \frown \langle x\rangle) \unlhd a \unrhd \|_{csi}^{\Phi}(\alpha \frown \langle y\rangle) \ \textit{otherwise}
\end{array}$$

for $\Phi \subseteq \Sigma$. If $\Phi = \emptyset$ then $\|_{csi}^{\Phi} = \|_{csi}$. The auxiliary deadlock at termination operator S_D which turns termination into deadlock, is defined as follows.

$$\begin{array}{ll}
S_D(S) & = D \\
S_D(D) & = D \\
S_D(x \unlhd a \unrhd y) & = S_D(x) \unlhd a \unrhd S_D(y)
\end{array}$$

4 Noninterference and behavioral noninterference

Informally speaking, a program behavior P is *noninterferent* (NI) if its low output does not depend on its high input. Formally, a process $P \in \mathit{NI}$ if for all inputs $v_1, v_2 \in V$ such that $v_1|_{V_L} = v_2|_{V_L}$,

$$(P \bullet v_1)|_{V_L} = (P \bullet v_2)|_{V_L}.$$

Now we define *behavioral noninterference* (BNI) for processes in BPPA. This definition essentially says that a process is secure if its actions are secure, and it cannot be interfered with implicit actions. In other words, its behavior from the view of non-implicit actions, is always the same whatever the returned boolean value after the execution of an implicit action is. In other words, its behavior from the view of non-implicit actions, is always the same whatever the returned boolean value after the execution of an implicit action is. Thus, if we rename all implicit actions of this process, then the obtained process must be branching bisimilar to its abstraction of implicit actions. Formally, a polarized process $P \in \mathit{BNI}$ if $\sigma(P) \subseteq \Sigma_s$ and $t_H(P) \leftrightarrows_b \tau_H(P)$.

5 Some results of our work

In this section, we present a few results of our work.

Theorem 1. *(***Soundness***) Let P be a polarized process. Then $P \in BNI \Rightarrow P \in NI$.*

Proof. This follows from the fact that $(P \bullet v)|_{V_L} = \tau_H \bullet v|_{V_L}$ for all $v \in V$.

Theorem 2. *(***Separability***) Let P_i $(1 \leq i \leq n)$ be polarized processes such that $var(P_i) \cap var(P_j) = \emptyset$ for all $i, j \in [1..n]$ and $i \neq j$, Then for every $\Phi \subseteq \Sigma$,*

$$\forall i (1 \leq i \leq n) : P_i \in NI \Rightarrow \|^{\Phi}_{csi} (\langle P_1 \rangle \curvearrowright \langle P_2 \rangle \curvearrowright \cdots \curvearrowright \langle P_n \rangle) \in NI.$$

Proof. This follows from the assumption that if $V = [V_1, \ldots, V_n]$ and $var(a) \subseteq var(V_i)$ for some i then $effect_a([v_1, \ldots, v_n]) = [v_1, \ldots, effect_a(v_i), \ldots, v_n]$ and $y_a([v_1, \ldots, v_n]) = y_a(v_i)$.

Theorem 3. *(***Congruence***) Let P_i and Q_i $(1 \leq i \leq n)$ be non-divergent polarized processes such that $P_i \leftrightarrow_b Q_i$. Then for any Φ such that $t \in \Phi$:*

$$\|^{\Phi}_{csi} (\langle P_1 \rangle \curvearrowright \langle P_2 \rangle \curvearrowright \cdots \curvearrowright \langle P_n \rangle) \leftrightarrow_b \|^{\Phi}_{csi} (\langle Q_1 \rangle \curvearrowright \langle Q_2 \rangle \curvearrowright \cdots \curvearrowright \langle Q_n \rangle).$$

Proof. Let $\mathcal{B}$ be a binary relation defined as follows. For non-divergent processes P and Q, $(P, Q) \in \mathcal{B}$ if there are non-divergent sequences α and β with the same length n for some $n \in \mathbb{N}$ such that $P = \|^{\Phi}_{csi} (\alpha)$, $Q = \|^{\Phi}_{csi} (\beta)$, and for all i: $\alpha_i \leftrightarrow_b \beta_i$. It is not hard to see that $\mathcal{B}$ is a branching bisimulation.

Theorem 4. *(***Compositionality***) Let P_i $(1 \leq i \leq n)$ be polarized processes such that $t_H(P_i)$ are non-divergent. Then for any $\Phi \supseteq H$,*

$$\forall i (1 \leq i \leq n) : P_i \in BNI \Rightarrow \|^{\Phi}_{csi} (\langle P_1 \rangle \curvearrowright \langle P_2 \rangle \curvearrowright \cdots \curvearrowright \langle P_n \rangle) \in BNI.$$

Proof. This follows from Theorem 3.

Proposition 1. *(***Decompositionality***) Let P_i $(1 \leq i \leq n)$ be finite polarized processes in which the termination is either S or D. Then for any $\Phi \supseteq H$,*

$$\|^{\Phi}_{csi} (\langle P_1 \rangle \curvearrowright \langle P_2 \rangle \curvearrowright \cdots \curvearrowright \langle P_n \rangle) \in BNI \Rightarrow \forall i (1 \leq i \leq n) : P_i \in BNI.$$

Proof. This can be proven by induction on the length of processes.

5.1 Concluding remarks and future works

We have addressed a new approach to verify noninterference for sequential and multi-threaded languages. Our definitions of noninterference have been defined in the setting of BPPA and Thread Algebra. We have shown that these definitions satisfy certain desirable properties. Since the definition of behavioral noninterference seems to be strict, we are working on some loose variants of BNI. Furthermore, our work can be extended to the work on recent object-oriented programming languages such as C♯ or Java. Moreover, we only cover the extension of the simplest interleaving strategy called *cyclic interleaving* of [3]. Other plausible interleaving strategies can also be adapted the feature of the cyclic interleaving with persistence operator defined in this paper.

References

1. G. Barthe, P.R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *IEEE Computer Security Foundations*, volume 17, pages 100–114, 2004.
2. D.E. Bell and L.J. La Padula. Secure computer system: mathematical foundations and model. Tech. Rep. M74-244, MITRE Corporation, Bedford, Massachussets, 1973.
3. J. Bergstra and C.A. Middelburg. Thread algebra for strategic interleaving. Computing Science Report 04-35, Department of Mathematics and Computing Science, Eindhoven University of Technology, 2004.
4. J.A. Bergstra and M.E. Loots. Program algebra for sequential code. *J. Logic Algebr. Programming*, 51:125–156, 2002.
5. G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Theoretical Computer Science*, 281:109–130, 2002.
6. D.E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.
7. J. Goguen and J. Meseguer. Secure policies and security models. In *IEEE Symp. on Security and Privacy*, pages 11–20, 1982.
8. J.L. Jacob. Separability and the detection of hidden channels. *Information Processing Letters*, 34:27–29, 1990.
9. R. Joshi and K. Leino. A semantics approach to secure information flow. *J. of Sci. Comput. Programming*, 37:113–138, 2000.
10. A.C. Meyers. Jflow: Practical mostly-static information flow control. In *ACM Symp. on Principles of Programming Languages*, pages 228–241, 1999.
11. A. Sabelfeld and A. Myers. Language-based information flow security. *IEEE J. on Selected Areas in Communications*, 21(1):5–19, 2003.
12. G. Smith and D. Volpano. Secure information flow in multi-threaded imperative languages. In *POPL'98*, volume 29, pages 355–364, 1998.
13. D. Volpano and G. Smith. A type approach to program security. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621, 1997.
14. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *J. of Computer Security*, 4(3):167–187, 1996.

Real-time system verification techniques based on abstraction/deduction and model checking

EunYoung Kang

LORIA-INRIA, France
Technical University of Delft, The Netherlands
Eun-Young.Kang@loria.fr

Abstract. Our research focuses on verification techniques for real-time systems based on predicate abstractions. These techniques aim to combine abstract interpretation, model checking, and theorem proving in order to obtain a powerful and highly automatic verification environment for real-time systems. One drawback of current real-time model checking approaches is the limited size of the systems that can be analyzed. For the computation of finite abstractions in the way of infinite-state systems analysis, we propose an Iterative-Abstract-Refinement algorithm. Using our algorithm, we can reduce the aforementioned drawbacks associated with the application of real-time model checking such as the limited applicability due to state space explosion characteristics

1 Introduction

The automatic verification problem for finite-state real-time systems has been considered and solved [1, 2, 16]. In many cases, theoretically optimal algorithms are known [4, 18]. Unfortunately, even if these algorithms are fully automatic, they are confronted with the state-explosion problem. They are typically exponential in the number and maximum values of clocks.

On the other hand, deductive techniques can in principle be used to verify infinite-state systems, based on suitable sets of axioms and inference rules. Although they are supported by theorem provers and interactive proof assistants, their use requires considerable expertise and tedious user interaction.

Abstract interpretation [8] provides a different approach to computing finite-state abstractions. For instance, predicate abstraction [10, 17] is a well known approach; given a transition system and a finite set of predicates, this method determines a finite abstraction, where each state of the abstract state space is a truth assignment to the abstraction predicates.

Model checking and abstraction/deductive techniques are therefore complementary. We propose an efficient scheme by combinations of those approaches that should give rise to powerful verification environments. For example, a theorem prover can be used to verify that a finite-state model is a correct abstraction of a given system, and properties of that finite-state abstraction can then be established using model checking. It relies on the fact that most properties can be shown correct without the need to maintain precise timing information for the system. A complex set of timed states can be safely abstracted by a simpler (abstract) one.

In general, the relationship between concrete and abstract models that underlies abstract interpretation is described by a Galois connection. The abstract domain is that Boolean lattice whose atoms are the set of predicates true or false of a set of states in a concrete model. The model obtained by abstraction w.r.t. this lattice exhibits at least behaviours of the concrete system; it may also include some bahaviours that have no counterpart in the concrete system.

The idea is to reduce the number of states of a model by abstracting away behaviours not essential to the verification. The genetic techniques are known as incremental abstraction refinement [6] and counterexample-guided abstraction refinement [7]. We have studied and partially formalised a variant that combines several techniques [7, 6], with some modifications. Abstraction can be constructed manually to exploit the structure of the model under consideration.

The idea is to perform an initial over-approximating abstraction of the model and then check for two requirements: (1)-the *conformance* between an abstract model and its concrete model, and (2)-the required property. If the abstract model conforms to its concrete model and the properties of interest can be successfully verified (or a positive result) over the abstract model, they also hold of the concrete system as well.

Otherwise, either the abstract model is not the correct representation of its concrete model or a negative result may be spurious (caused by extra states added during the over-approximation). In this case, we iteratively concretize (or refine) the abstract model in order to construct stronger invariants and rule out some of their extra behaviours until a positive or true negative result is obtain. We call such a concretization process an *abstraction-refinement*, and we call the model resulting from the *abstraction-refinement* a *complete model*

Our idea for advances in the size of systems that can be analyzed by using several abstraction methods is not the first. A comparison with similar work on abstract interpretations, approximations for real-time systems [9] and predicate abstractions appears in the section 3.

2 Approach and proposed solution

We have proposed a tool supported methodology based on the combination of abstract/deductive and real-time model checking techniques. In order to make our approach more concrete, we present one algorithm, called *Iterative-abstract-refinement algorithm* (IRA) to verify a rich class of safety and liveness properties of a timed system based on computing a finite abstraction of the system by succ essive *abstraction-refinement*.

Figure 1 shows models over an abstract, a complete, and a concrete domain. The abstract model shown in Fig.1.(a) cannot guarantee whether the abstract model-(a) is able to verify given properties and preserve every possible behaviour of a concrete model-(c). The model of Fig.1.(b) has been obtained by *abstraction-refinement*; it is *complete* for verifying the properties of the concrete model-(c).

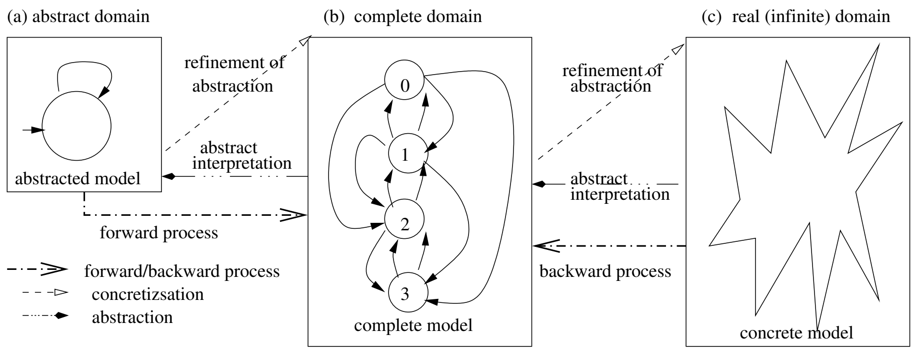


Fig. 1. Abstract, complete, and concrete models

We expect to obtain a *complete* model-(b) from an abstract model-(a) using IRA.

In this paper, we will use eXtended Timed automata Graphs [3] as the formalism to represent (concrete) timed system-(c), and Predicate Diagrams for Timed systems [14] as the way of presenting predicate abstraction for XTG.

The input to IRA consists of three parameters; the XTG representing the concrete system, the property to be verified, and a finite set of predicates to be used for refinements. For the sake of efficiency we require predicates (Boolean expressions) over a set of *configurations* of XTG that are constraints, invariants, guards, locations and transitions of XTG.

Our methodology is based on *abstraction-refinement* framework. IRA has two (forward/backward) processes. An initial abstraction PDT can be obtained from the XTG by a backward step. This backward step (process) is performed by selecting a set of XTG configurations and considering them as predicates of PDT. Starting from the initial PDT, IRA iterates forward process until PDT is *complete*:

Backward process: direction (c) to (a) A trivial and potentially incomplete PDT is extracted from XTG by Boolean abstraction. Its atoms are the subset of given predicates over a set of XTG configurations and properties of interest as well.

Forward process: direction (a) to (b) IRA picks a PDT and – if the PDT appears to be complete – infers successful verification and preservation, or – if the PDT is not complete – enforces completeness by two operations: a *splitting* operation and an *excluding* operation

1. A *splitting operation* is done while IRA checks correctness of preservation in the way of *conformance* checking between a PDT and XTG.
   – If IRA fails to prove *conformance* then IRA does splitting the PDT w.r.t predicates in order to enrich the PDT and also to add details in the PDT.

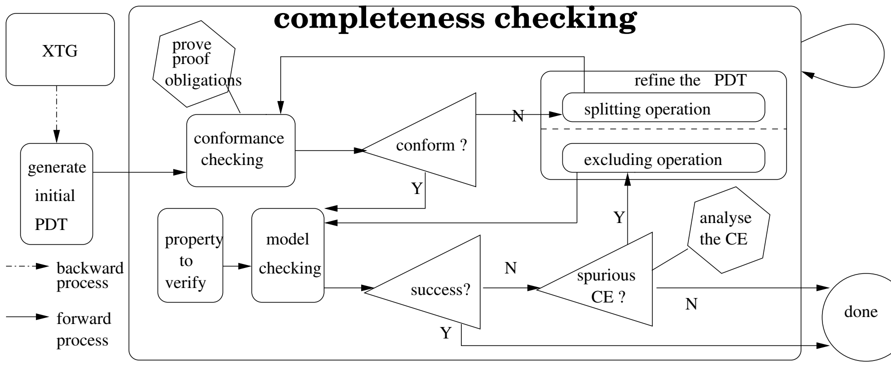


Fig. 2. Overview of IRA

  - During the operation, a set of proof obligations (a number of verification conditions expressed in first-order logic) are proved in order to eliminate extra duplicated relations (paths) among PDTs caused by splitting. This is continued until the PDT conforms to XTG.
2. An *excluding operation* is done after model checking (or after *conformance* checking).
  - If we limit ourselves to univeral properties, then if a property fails in the PDT we can generate a counterexample trace in the PDT and attempt to find a corresponding concrete trace in XTG.
  - If one exists then the property is false in XTG and the verification fails (true negative result is obtain). Otherwise the abstract counterexample is spurious and abstraction is too coarse so we refine it by *excluding* some abstracted bogus transition-relations that are not present in XTG (found by analysing the counterexample).
  - We then recheck the property. This is continued until either the property is verified or a concrete conterexample found.

Figure 2 shows the overall framework. The above approach is tested in [14] and partially validated. The main contribution up to now is that we have identified a suitable format that serves as an interface between deductive and model checking techniques, intended for the verification of real-time systems. We also have established a set of verification conditions that are sufficient to prove *conformance* between PDT and XTG.

However, it still lacks automation both in the computation of abstraction and in identifying the predicates for splitting. Moreover, We have not fully validated the above approach, for instance, our experiment in [14] does not consider a spurious counterexample trace upon failure during model checking procedure, thus any counterexample-guided abstraction refinement method does not really used during the execution of the *abstraction-refinement* procedure. Considering such current weakpoints and comparing with other related work in the next section, we will discuss future work at the end of this paper.

3 Related work

The techniques described in this paper can be viewed in an abstract interpretation sense as a combination of abstraction, operation on an abstract domain and concretization. Our refinement bears resemblance to refinement as in the B method [6]. It allows one to enrich a model in a step by step approach. Refinement provides a way to construct stronger invariants and also to add details in a model. It is also used to transform an abstract model into a more concrete version by modifying the state description.

Halbwachs [11] successfully applies abstract interpretation to synchronous reactive systems as a way of state space exploration. But he does not consider abstractions over control information (only data information is abstracted). Dill and Wong-Toi [9] use both over- and under-approximations as abstractions, and for finite-state systems, automatically determine whether there are reachable violating sates. Their refinements are different than ours. They refine (over-approximations only) the set of reachable states on paths to violating states. However their techniques are limited to proving invariants.

Predicate abstraction has emerged as a fruitful basis for software verification. Based on predicate abstraction, Namjoshi and Kurshan [15] compute finite bisimulations of timed automata. However, currently it is unclear whether their approach is applicable in practice.

Our basic assumption undeliying predicate abstraction is that for the verification of a given property, the state space of an XTG can be partitioned into finitely many equivalence classes. For example, the precise amount of time elapsed in a transition does not really matter as long as the clock values are within certain bounds and similarly, the precise values of the data can be abstracted with the help of predicates that indicate characteristic properties.

Predicate abstraction also underlies tools such as SLAM [5] and BLAST [12] that compute abstraction refinements on the basis of spurious counter-examples provided by model checking. They refine abstractions in such a way that the spurious counter-example is avoided.

In symbolic model checking for real-time systems, difference bound matrices (DBM) are used to represent a set of state spaces (regions) over real variables. The representation we use is rather tailored to IRA approaches, since it is able to efficiently deal with the two (*splitting/excluding*) operations required for such approaches. For those operations it is not necessary to have a canonical model available. In IRA, the represented region is the consequence of repeated *abstraction-refinement* by splitting PDT and excluding its abstracted bogus transition-relations.

Our early work on combining tools for abstract interpretation and state space exploration has been reported in [13]. However, extra steps used in the algorithm proposed there often fail to significantly reduce the state space.

4 Further work

In IRA, the predicates are assumed to be given by the user, or they are extracted syntactically from the system description. It is obviously difficult for us to find the right set of predicates. We are investigating further heuristics to be able to discover automatically all the needed predicates for the practical algorithm.

We abstract from the precise amount of time that may elapse in a time-passing transition. Thus, we cannot easily verify properties that describe the timing behavior of a system. We intend to study two possible solutions to this problem, either by using a timed temporal logic (TLTL) or by introducing auxiliary clocks during verification. In any case, we would want to take advantage of model checking tools for real-time systems.

Besides, we aim at reducing the number of verification conditions that users have to discharge with the help of a theorem prover in order to establish *conformance*. It will be interesting to restrict attention to specific classes of systems that give rise to decidable proof obligations.

We are also interested in adding counterexample-guided abstraction refinement method in SLAM and BLAST to IRA and validating a full scheme proposed in Figure 2.

Although IRA for now shows that we have not reached the ideal combining algorithm yet, it clearly helps in identifying opportunities for proper incorporation of abstraction/deduction and model checking for real-time systems in practical situations.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104:2–34, 1993.
2. R. Alur and D. Dill. The theory of timed automata. *Theoretical Computer Science*, (126):183–235, 1994.
3. M. Ammerlaan, R. Lutje Spelberg, and W.J. Toetenel. XTG – an engineering approach to modelling and analysis of real-time systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 88–97. IEEE press, 1998.
4. Tobias Amnell and many others. UPPAAL: Now, next, and future. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, LNCS(2067):99-124. Springer-Verlag, Berlin, 2001.
5. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Principles of Programming Languages (POPL 2002):1–3*, 2002.
6. Dominique Cansell and Dominique Mery. Tutorial on the event-based B method. Technical report, LORIA-INRIA, 2004.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *12th CAV00, LNCS(1855):154–169*. Springer-Verlag, 2000.
8. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2-3):103–179, 1992.
9. D. Dill and H. Wong-Toi. Verification of real-time systems by successive over and under approximation. In *7th CAV95, LNCS(939):409–422*. Springer-Verlag, 1995.

10. S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *9th CAV97, LNCS(1254):72–83*. Springer-Verlag, 1997.
11. N. Halbwachs. Delay analysis in synchronous programs. In *CAV93, LNCS(697)*. Springer-Verlag, 1993.
12. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth McMillan. Abstractions from proofs. In *31st POPL*. ACM Press, 2004.
13. EunYoung Kang. Parametric analysis of real-time embedded systems with abstract approximation interpretation. In *26th ICSE*, 2004.
14. EunYoung Kang and Stephan Merz. Predicate diagrams for the verification of real-time system. In *5th AVoCS05*, ENTCS, 2005.
15. K. Namjoshi and R.Kurshan. Syntactic program transformations for automatic abstraction. LNCS(1855):435–449, 2000.
16. S.Tripakis. *The Formal Analysis of Timed Systems in practice*. PhD thesis, University of Joseph Fourrier de Grenoble, 1998.
17. Y.Kesten and A.Pnueli. Modularization and abstraction: The keys to practical formal verification. In *23th MFCS98, LNCS(1450):54-71*. Springer-Verlag, 1998.
18. S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1997.

Discrete Simulation of Behavioural Hybrid Process Calculus

Tomas Krilavičius[⋆] and Helen Schonenberg

Formal Methods and Tools, Univ. of Twente, 7500AE Enschede, The Netherlands,
(T.Krilavicius, M.H.Schonenberg)@cs.utwente.nl,
WWW home page: http://www.cs.utwente.nl/~krilaviciust

Abstract Hybrid systems combine continuous-time and discrete behaviours. Simulation is one of the tools to obtain insight in dynamical systems behaviour. Simulation results provide information on performance of system and are helpful in detecting potential weaknesses and errors. Moreover, the results are handy in choosing adequate control strategies and parameters.
In our contribution we report a work in progress, a technique for simulation of Behavioural Hybrid Process Calculus, an extension of process algebra that is suitable for the modelling and analysis of hybrid systems.

1 Introduction

The growing interest in hybrid systems both in computer science and control theory has generated a new interest in models and formalisms that can be used to specify and analyse such systems. A prominent framework for hybrid systems is provided by the family of hybrid automata models (hybrid automata [1], hybrid behavioural automata [2], hybrid input/output automata [3]). More recently process algebraic models have been put forward as a vehicle for the study of hybrid systems [4,5,6,7].

Simulation is a *de facto* standard tool in both academia and industry for analysis of hybrid systems. It helps to detect potential weaknesses and errors, and provides information on performance of system. There is a number of simulation tools which provide various facilities for analysis of hybrid systems. Hybrid χ [6] provides facilities for simulation of hybrid process calculus. HyVisual [8] is a Java based visual modeller and simulator for hierarchical continuous-time dynamical and hybrid systems. Dymola[1], Stateflow/Simulink [9] and 20-Sim[2] provide industrial strength facilities for simulation of non-causal object oriented simulation language Modelica™ [10], hierarchical formalism and bond graphs [11], respectively.

[⋆] Work is partially done in the framework of the HYCON Network of Excellence, contract number FP6-IST-511368*.

[1] See http://www.Dynasim.se.

[2] http://www.20sim.com/

We report a work in progress, a technique for simulation of Behavioural Hybrid Process Calculus (BHPC) [7]. BHPC is a process calculus that extends the standard repertoire of operators that combine discrete functional behaviour with features to also represent and compose continuous-time behaviour. Dynamic behaviour is represented by the evolution of variables, which are typically defined in terms of differential equations. Following [12], behaviour can be simply seen as the set of all allowed real-time evolutions, or *trajectories*, of the system variables. The operational semantics of the calculus defines the transitions for the simulator. An adapted version of the expansion law from [13] is used to solve parallel composition. As a first step towards simulation of entire BHPC, we propose a discrete simulator. It abstracts from the continuous-time behaviour and uses operational semantics rules and the expansion law to determine the next simulation step. We design it in such a way that it should be easy extendible to hybrid simulation.

2 Behaviour Hybrid Process Calculus

In this section we introduce main concepts of Behavioural Hybrid Process Calculus. See [7] for the details and proofs.

Trajectories. The continuous behaviour of hybrid systems can be seen as the set of continuous-time evolutions of system variables. We will call them *trajectories*. We assume that trajectories are defined over bounded time intervals $(0, t]$, and map to a *signal space* to define the evolution of the system. The signal space ($\mathbb{W}$) specifies the potentially observable continuous behaviour of the system. Components of the signal space correspond to the different aspects of the continuous behaviour, like temperature, pressure, etc. They are associated with *trajectories qualifiers* that identify them.

Hybrid Transition System. We define a hybrid transition system as a collection $HTS = \langle S, A, \rightarrow, W, \Phi, \rightarrow_c \rangle$, where S is a *state space*. The *discrete transition relation* $\rightarrow \subseteq S \times A \times S$ defines discrete changes annotated by actions ($\mathsf{a} \in A$). The *continuous-time* transition relation $\rightarrow_c \subseteq S \times \Phi \times S$ links continuous changes to trajectories ($\varphi \in \Phi$).

Language. We introduce a language for defining hybrid processes.

$$B ::= \mathbf{0} \mid \mathsf{a} \,.\, B \mid [\varphi] \,.\, B \mid \bigoplus_{i \in I} B_i \mid B \parallel_A^H B \mid P$$

Action-prefix $\mathsf{a}.B$ defines a process that starts with action a and afterwards engages in B. *Silent actions*[13] (denoted τ) are used to specify nondeterministic behaviour.

Trajectory-prefix $[\varphi].B$ models the behaviour of a process that executes a continuous trajectory φ and then continues as B.

Superposition $\bigoplus_{i \in I} B_i$ is a generalised operator on sets of behavioural expression. For action prefixes the interpretation of the operator is the same as the ordinary choice operator Σ from classical process algebras. However, the choice among trajectories is made at the moment when the trajectories start bifurcating.

Parallel composition $B_1 \parallel_A^H B_1$ specifies the behaviour of two parallel processes. The operator explicitly attaches the sets of synchronising action names A and of synchronising trajectory qualifiers H. Synchronisation on actions has an interleaving semantics. Trajectory-prefixes can evolve in parallel only if the evolution of coinciding trajectory qualifiers is equal.

Recursion allows to define processes in terms of each other, as in the equation $B \triangleq P$, where B is the process identifier and P is a process expression that may only contain actions and signal types of B.

One of the main tools to compare systems is a *strong bisimulation.* The bisimulation for continuous dynamical systems is presented in [14]. The process algebraic version is discussed in [13]. A strong bisimulation for hybrid transition systems requires both systems to be able to execute the same trajectories and actions and to have the same branching structure. Strong bisimulation for BHPC is a congruence relation w.r.t. all operations defined above. See [7] for details.

BHPC is an assembly language for a modelling of hybrid systems. We add auxiliary constructs to increase usability of the language.

We introduce *parametrised action-prefix* $\mathsf{a}_v \,.\, B(v) \triangleq \bigoplus_{v \in V} \mathsf{a}_v \,.\, B(v)$ for convenience (as in [13, 53–58]).

We will use a *symbolic trajectory-prefix*, which extends a notion of ordinary trajectory-prefix by providing a set of continuous behaviours conforming to the certain conditions. We will define a *set of trajectory-prefixes* $[f \mid \Phi] \,.\, B(f) \triangleq \bigoplus_{\varphi \in \Phi} ([\varphi] \,.\, B(\varphi))$ where Φ is a set of trajectories and f is a *trajectory variable* for a trajectory. Furthermore, we will use $[t_1, \ldots, t_m \mid \Phi \downarrow \mathcal{P}red \Downarrow \mathcal{P}red_{\text{exit}}]$ to define extended version of set of trajectory-prefixes, where $t_1, \ldots, t_m$ are trajectory qualifiers, which can be used to access corresponding parts of trajectories. Two types of restrictions on the set of trajectories are used: $\downarrow$ states restrictions on the whole duration of trajectories and $\Downarrow$ define the *exit conditions*, i.e., restrictions on the *end-points.*

Sometimes it is useful to check some conditions explicitly, and if they are not satisfied, to stop the progress of process. With the *guard* construct $\langle \mathcal{P}red \rangle \,.\, B$, these conditions can be given as a predicate.

Idling in BHPC is defined as $\mathsf{idle} = \left[t \;\middle|\; \dot{t} = 0\right]$, where t is a reserved variable. It does not manifest any observable behaviour, but reacts as soon, as it is invoked by another process, which communicates with the process, which follows the idling period.

Application of BHPC. Bouncing ball [7] is a simplified model of an elastic ball that is bouncing and losing a fraction of its energy with every bounce. The altitude of the ball is h, and v is a vertical speed, c is a coefficient for the lost energy. The ball moves according to the flow conditions and at the bounce time the variables are reassigned. In BHPC it can be defined in the following way:

$$\mathrm{BB}(h_0, v_0) \triangleq \big[h, v \mid \Phi(h_0, v_0) \Downarrow h = 0\big] \,.\, \mathrm{BB}(0, -c * v)$$
$$\Phi(h_0, v_0) = \{h, v : (0, t] \to \mathbb{R} \mid h(0) = h_0, v(0) = v_0, \dot{h} = v, \dot{v} = -g, h \geqslant 0\}$$

Symbolic trajectory-prefix $\big[h, v \mid \Phi(h_0, v_0) \Downarrow h = 0\big]$ defines the dynamics of the ball until the bounce, and then the process continuous recursively calling itself with updated continuous variables $\mathrm{BB}(0, -c * v)$. We extend the given specification by adding discrete actions to sense the elasticity of the bounce and increase the ball's kinetic energy, and a compensating controller (CC).

$$\begin{aligned}
\mathrm{BB}(h_0, v_0) \triangleq\ & \big[h, v \mid \Phi(h_0, v_0) \Downarrow h = 0\big] \,.\, \mathsf{bounce}(c : [0,1]). \\
& \big[h, v \mid \Phi(0, -cv) \Downarrow v = 0\big] \,.\, \mathsf{push}(v : \mathbb{R}) \,.\, \mathrm{BB}(h, v) \\
\mathrm{CC}(v_0) \triangleq\ & \mathsf{idle} \,.\, \mathsf{bounce}(c : [0,1]) \,.\, \mathsf{idle} \,.\, \mathsf{push}\left((1-c)\, v_0\right) \,.\, \mathrm{CC}\left((1-c)\, v_0\right) \\
\mathrm{Sys}(h_0, v_0) \triangleq\ & \mathrm{BB}(h_0, v_0) \parallel^{v}_{\mathsf{push},\mathsf{bounce}} \mathrm{CC}(v_0)
\end{aligned}$$

3 Simulating BHPC

As the first step, we will focus on the simulation of discrete behaviour of BHPC. Discrete simulation can give a lot of valuable insight on the system. It helps to detect potential deadlocks. Discrete abstraction of the system can be verified and error-traces can be used in an hybrid simulator to investigate potential faults. Moreover, it comes handy in the early stages of modelling or prototyping.

To get discrete abstraction we make some choices concerning continuous-time behaviours. We discuss diverse choices for the operators individually.

– Parametrised action-prefix is left unchanged, only the parameters related with trajectories are ignored.
– We will interpret (symbolic) trajectory-prefix as a special type of action-prefix. Several options are possible. It can be seen as a silent (unobservable) action, just like the τ action described in [13]. A special action can be introduced to denote any trajectory-prefix. Then trajectory-prefixes can be treated like ordinary action-prefixes.
– Guard is treated as usually, but with all trajectories-related predicates evaluated to `true` or `false`, depending on the simulation purpose.
– With the trajectory-prefix reduced to a discrete action, the superposition operator is equal to the choice operator from ordinary process algebra.
– Without the set of synchronising trajectory qualifiers, the parallel composition operator from BHPC becomes equal to the parallel composition operator from ordinary process algebra.

Operational semantics for the language already define the transitions the simulator can take, only parallel composition requires additional reshaping. It is provided by the expansion law [7], which expresses parallel composition as a superposition of processes.

4 Future Plans

Future plans for the simulation of BHPC and the calculus include several directions.

The discrete simulator is being designed in such a way that extending it to the hybrid simulation should be doable without completely reshaping the discrete simulation part. Principally, it means adding support for continuous-time behaviour: interpretation of trajectory qualifiers, interface to solvers, etc.

Moreover, we are building the current tool not as a prototype of an industrial tool, but more as a hybrid *"sand-box"*, a place to experiment with BHPC and related developments. Consequently, the architecture and implementation of the tool are being designed in such a way that it is easy to accommodate the changes in the calculus and to test the algorithms developed for hybrid systems in BHPC framework. Adaptable and well documented interfaces for ODE/DAE solvers should be provided for experimenting with different approaches for continuous-time behaviour simulation. Co-simulation also takes place in the plans. Theoretical and practical issues of the co-simulation of BHPC and Simulink[3] are explored as a part of WP3 of HYCON[4]. Furthermore, the means to export a restricted subset of BHPC to Modelica™ [10] are investigated.

Examination of different simulation results visualisation techniques are of interest too. Just ordinary graphs (plots) usually used to display continuous-time simulation results do not provide sufficient information about switching behaviour. Event-traces and the message sequence charts [15] are appropriate and even beneficial when discrete behaviour should be visualised. However both techniques become inadequate when combination of continuous-time and discrete behaviour should be visualised. It calls for a combination of the aforementioned techniques or even new unconventional approaches.

One more interesting development of BHPC is *model-based testing* of hybrid systems [16]. In model-based testing we use a formal specification (a model) of the desired behaviour of the system under test (SUT). The model is used to select the input for the SUT. The model is also used to check the correctness of the output of the SUT after a certain input. One of the great advantages of model-based testing is that tools can explore the model and automatically generate and execute tests cases from the model. BHPC and the simulation tool can be extended to generate required information for a well-known on-the-fly testing tool TorX [17], and in combination with it form a hybrid systems testing framework.

5 Results

In this paper we proposed a simulation technique for Behavioural Hybrid Process Calculus. The calculus and transitions system were introduced, operators for the

[3] `http://www.mathworks.com/products/simulink/`.
[4] WP3, HYCON, `http://wp3.hycon.bci.uni-dortmund.de/`.

calculus were explained. We focussed discussion on the discrete simulation of selected operators, by abstracting from the continuous-time behaviour so that all operators from our calculus have corresponding interpretation in ordinary process algebra. The expansion law is used to resolve parallel composition. The operational semantics defines the possible transitions for the simulator.

The work in progress will have to evaluate the conceptual and practical implications of our approach. Currently we are developing techniques and tools for discrete and hybrid simulation of the calculus.

Acknowledgements. The authors thank H. Brinksma for his comments.

References

1. Alur, R., Courcoubetis, C., Henzinger, T., Ho, P.H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Grossman, R.L., Nerode, A., Ravn, A.P., Rischel, H., eds.: Hybrid Systems. Volume 736 of LNCS., Springer (1993) 209–229
2. Julius, A.: On Interconnection and Equivalence of Continuous and Discrete Systems: A Behavioral Perspective. PhD thesis, SSCG, Univ. of Twente (2005)
3. Lynch, N., Segala, R., Vaandrager, F.: Hybrid I/O automata. Information and Computation **185** (2003) 105–157
4. Cuijpers, P.J.L., Reniers, M.A.: Hybrid process algebra. Technical report, Dept. of Math. and Comp. Science, Tech. Univ. of Eindhoven (TU/e), Eindhoven (2003)
5. Bergstra, J., Middelburg, C.: Process algebra for hybrid systems. Technical report, Dept. of Math. and Comp. Science, Tech. Univ. of Eindhoven (TU/e), Eindhoven (2003)
6. van Beek, D., Man, K., Reniers, M., Rooda, J., Schiffelers, R.: Syntax and consistent equation semantics of hybrid chi. Report CS-Report 04-37, Tech. Univ. of Eindhoven (TU/e), Eindhoven (2004)
7. Brinksma, E., Krilavičius, T.: Behavioural hybrid process calculus. Technical Report TR-CTIT-05.45, CTIT, University of Twente (2005)
8. Lee, E., Zheng, H.: Operational semantics of hybrid systems. In: Hybrid Systems: Computation and Control. LNCS (2005) 25–53
9. Hamon, G., Rushby, J.: An operational semantics for Stateflow. In Wermelinger, M., Margaria-Steffen, T., eds.: FASE 2004. LNCS (2004) 229–243
10. Modelica Association: Modelica - A Unified Object Oriented Language for Physical Systems Modeling: Language Specification. (2005)
11. van Amerongen, J., Breedveld, P.: Modelling of physical systems for the design and control of mechatronic systems. Annual Reviews in Control **27** (2003) 87–117
12. Polderman, J., Willems, J.C.: Introduction to Mathematical Systems Theory: a behavioral approach. Springer (1998)
13. Milner, R.: Communication and concurrency. Prentice-Hall, Inc. (1989)
14. van der Schaft, A.: Bisimulation of dynamical systems. In Alur, R., Pappas, G.J., eds.: HSCC. Volume 2993 of LNCS., Springer (2004) 555–569
15. ITU-T: Recommendation Z.120. Message Sequence Charts. Technical Report Z-120, Int. Tel. Union, Genève (2000)
16. Berkenkötter, K., Kirner, R.: Real-Time and Hybrid Systems Testing. In: Model-based Testing of Reactive Systems. Volume 3472 of LNCS. Springer (2005) 355–387
17. Bohnenkamp, H., Belinfante, A.: Timed testing with TorX. In: Formal Methods Europe. Volume 3582 of LNCS., Springer (2005) 173 – 188

Formal Verification of Chi models using PHAVer

K.L. Man, R.R.H. Schiffelers
Eindhoven University of Technology, Eindhoven, The Netherlands
{k.l.man, r.r.h.schiffelers}@tue.nl

Abstract

The hybrid Chi (χ) formalism is a formalism for modeling, simulation and verification of hybrid systems. One of the most widely known hybrid system formalisms is that of hybrid automata. The translation of χ to hybrid automata enables verification of χ specifications using existing hybrid automata based verification tools. In this paper, we describe the translation from χ to hybrid automata, and the relation between hybrid automata and the linear hybrid I/O automata that are used for the verification tool PHAVer (Polyhedral Hybrid Automaton Verifyer). In the case study, we translate a χ specification to a linear hybrid I/O automaton, and use PHAVer to verify properties.

1 Introduction

Hybrid systems represent a domain where the dynamics and control (DC) and computer science (CS) world views meet, and we believe that a formalism that integrates the DC and CS world views is a valuable contribution towards integration of the DC and CS methods, techniques, and tools. The hybrid χ formalism [1] is such a formalism. On the one hand, it can deal with continuous-time systems, piecewise affine (PWA) systems, mixed logic dynamical (MLD) systems, linear complementarity (LC), and hybrid systems based on sets of ordinary differential equations using discontinuous functions in combination with algebraic constraints (the DC approach). On the other hand, it can deal with discrete-event systems, without continuous variables or differential equations, and with hybrid systems in which discontinuities take place (mainly) by means of actions (the CS approach). The intended use of hybrid χ is for modeling, simulation, verification, and real-time control. Its application domain ranges from physical phenomena, such as dry friction, to large and complex manufacturing systems. The history of the χ formalism dates back quite some time. It was originally designed as a modeling and simulation language for specification of discrete-event, continuous-time or combined discrete-event/continuous-time models. The first simulator [2], however, was suited to discrete-event models only. The simulator was successfully applied to a large number of industrial cases, such as an integrated circuit manufacturing plant, a brewery, and process industry plants

[3]. Later, the hybrid language and simulator were developed [4, 5]. For the purpose of verification, the discrete-event part of the language was mapped onto the process algebra χ_σ by means of a syntactical translation. The semantics of χ_σ was defined using a structured operational semantics style (SOS), bisimulation relations were derived, and a model checker was built [6]. In this way, verification of discrete-event χ models was made possible [7]. The hybrid χ formalism defined in [1] integrates the modeling language and the verification formalism. It resulted in a formalism with the straightforward and elegant syntax and semantics that is also highly suited to non-computer scientists. In the remainder of this paper, we usually refer to hybrid χ as χ.

One of the most successful formalisms for hybrid system verification is the theory of hybrid automata. In [1], formal translations between χ and hybrid automata (in both directions) have been defined. The translation from hybrid automata to χ aims to show that the χ formalism is at least as expressive as the theory of hybrid automata. The translation from (a subset of) χ to hybrid automata enables verification of χ specifications using existing hybrid automata based verification tools.

This paper is organized as follows: Section 2 gives a short introduction of χ. Connections between χ and other formalisms from DC and CS can be found in Section 3. The translation from χ to hybrid automata is described in Section 4. The relation between hybrid automata and the linear hybrid I/O automata that are used for the verification tool PHAVer (Polyhedral Hybrid Automaton Verifyer) is discussed in Section 5. In Section 6, we translate a χ specification of a water-level monitor to a corresponding linear hybrid I/O automaton, and use PHAVer to verify properties. Finally, some concluding remarks are made in Section 7.

2 The hybrid χ language

In this section, we briefly introduce the χ language. The introduction presented here is adapted from [1].

A χ process is a triple $\langle p, \sigma, E \rangle$, where p denotes a process term, σ denotes a valuation, and E denotes an environment. A valuation is a partial function from variables to values. Syntactically, a valuation is denoted by a set of pairs $\{x_0 \mapsto c_0, \ldots, x_n \mapsto c_n\}$, where x_i denotes a variable and c_i its value. The environment E is a tuple (C, J, L, H, R), where C, J, L denote the set of continuous variables, the set of jumping variables, and the set of algebraic variables, respectively, H is a set of channels, and R denotes a recursive process definition (partial function from recursion variables to process terms). The valuation σ and the environment E, together define the variables that exist in the χ process and the variable classes to which they belong. In χ, there is the predefined variable $\mathsf{time} \in \mathrm{dom}(\sigma)$, that denotes the current (model) time.

Process terms are the 'core' elements of the χ formalism. The set of process terms P is defined by the following grammar for the process terms $p \in P$:

$$\begin{array}{rl} p ::= & W : r \gg l_{\mathrm{a}} \;\mid\; u \;\mid\; \perp \;\mid\; \delta \;\mid\; [p] \;\mid\; u \curvearrowright p \;\mid\; p;p \\ \mid & b \rightarrow \;\mid\; p \,[\!]\, p \;\mid\; p \parallel p \;\mid\; h\,!!\,\mathbf{e}_n \;\mid\; h\,??\,\mathbf{x}_n \;\mid\; \partial_A(p) \\ \mid & \upsilon_H(p) \;\mid\; X \;\mid\; \iota_{J^+}(p) \;\mid\; [\![_{\mathrm{V}}\, \sigma_\perp, C, L \mid p\,]\!] \;\mid\; [\![_{\mathrm{H}}\, H \mid p\,]\!] \\ \mid & [\![_{\mathrm{R}}\, R \mid p\,]\!] \end{array}$$

$p \in P$ consists of atomic process term: action predicate $W : r \gg l_{\mathrm{a}}$, delay predicate u, inconsistent process term $\perp$, deadlock process term δ, send process term $h\,!!\,\mathbf{e}_n$, receive process term $h\,??\,\mathbf{x}_n$; unary operators: the any delay operator [ ], signal emission operator $\curvearrowright$, guarded operator $\rightarrow$, encapsulation operator $\partial_A()$, urgent communication operator $\upsilon_H(\;)$, recursive definition X, jump enabling operator $\iota_{J^+}(\;)$, variable scope operator $[\![_{\mathrm{V}}\, \sigma_\perp, C, L \mid \;]\!]$, channel scope operator $[\![_{\mathrm{H}}\, H \mid \;]\!]$, recursion scope operator $[\![_{\mathrm{R}}\, R \mid \;]\!]$, and binary operators: sequential composition ; , alternative composition operator $[\!]$, and parallel composition operator $\parallel$.

A more detailed explanation of χ and the semantics of χ can be found in [1].

3 Connections between other formalisms and χ

As mentioned previously, one of the most important concepts of χ is the integration between the DC and CS world views. Affine systems are powerful tools for describing or approximating hybrid systems (DC world views). General translation schemes from continuous-time piecewise affine systems and discrete-time piecewise affine systems to χ are defined, which show that these formalisms are closely related.

On the other hand, the study of hybrid systems in computer science (CS world views) is mainly focussed on the theory of hybrid automata. Formal translation from the theory of hybrid automata to χ has been defined. The translation from hybrid automata to χ aims to show that the χ formalism is at least as expressive as the theory of hybrid automata.

All above-mentioned formal translations can be found in [1]. For illustration purposes, in this section, we only give the translation of continuous-time piecewise affine systems to χ, and the translation from hybrid automata to χ by means of the thermostat example.

3.1 Continuous-time PWA to χ

Continuous-time piecewise affine systems are described by N systems of affine differential equations

$$\begin{array}{rcl} \dot{x}(t) & = & A_i x(t) + B_i u(t) + f_i \\ y(t) & = & C_i x(t) + D_i u(t) + g_i \end{array} \qquad \text{for} \begin{bmatrix} x(t) \\ u(t) \end{bmatrix} \in \Omega_i,$$

where i $(i = 1, \ldots, N)$ is the number of the mode. Each mode i is defined in a region Ω_i, which is a convex polyhedron, given by a finite number of linear inequalities, in the input/state space. Here, $u(t) \in \mathbb{R}^m$, $x(t) \in \mathbb{R}^n$, and $y(t) \in \mathbb{R}^l$ denote the input, state and output, respectively, at time t. Furthermore, f_i,

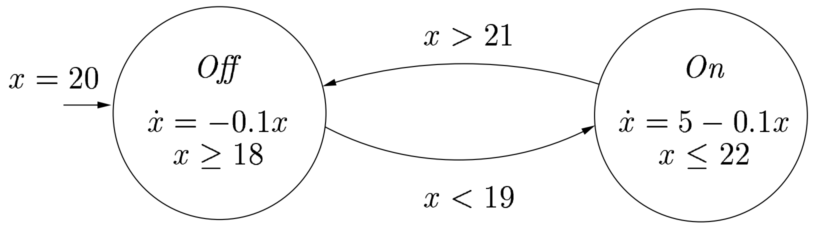


Figure 1: A hybrid automaton model of a thermostat

and g_i denote constants. In each mode, the trajectories of the state variables x are continuous functions of time. The trajectories of the input/output variables in a mode may be discontinuous functions of time.

Continuous-time PWA systems using the Caratheodory solution concept, can be translated to χ as follows:

$$
\begin{array}{l}
\langle\ \mathsf{cont}\ x,\ \mathsf{alg}\ y \\
,\ x = x_0 \\
\mid\ (\Omega_1 \Rightarrow \dot{x} = A_1 x + B_1 u + f_1,\ y = C_1 x + D_1 u + g_1) \\
\quad \wedge \\
\quad \vdots \\
\quad \wedge \\
\quad (\Omega_N \Rightarrow \dot{x} = A_N x + B_N u + f_N,\ y = C_N x + D_N u + g_N) \\
\rangle
\end{array}
$$

The state variables x are modeled by means of (non-jumping) continuous variables, with initial value x_0. The output variables y are modeled by means of algebraic variables. The behavior of u is not specified, as in the original PWA model. In the χ specification, u could denote a function of time, or u could be defined as an algebraic variable, and additional equations specifying the behavior of u could be added. The behavior associated to a mode i is described by means of a delay predicate $(\Omega_i \Rightarrow \dot{x} = A_i x + B_i u + f_i,\ y = C_i x + D_i u + g_i)$.

3.2 Hybrid automata to χ

This section shows the translation of a hybrid automaton model of a thermostat to χ. The hybrid automaton (adapted from [8]) is shown in Figure 1. Variable x represents the temperature. The control modes are *On* and *Off*.

Initially, the temperature equals 20 degrees, and the heater is off (control mode *Off*). The temperature falls according to the flow condition $\dot{x} = -0.1x$. According to the jump condition $x < 19$, the heater may go on as soon as the temperature falls below 19 degrees. The invariant condition $x \geq 18$ ensures that at the latest the heater will go on when the temperature equals 18 degrees. In the control mode *On*, the heater is on, and the temperature rises according to the flow condition $\dot{x} = 5 - 0.1x$. When the temperature rises above 21 degrees, the heater may turn off. Due to the invariant condition $x \leq 22$, at the latest the heater will turn off when the temperature equals 22 degrees.

This model is translated to χ specification (with some simplification), which results as follows:

$$
\begin{array}{llll}
\langle\ \textsf{cont}\ x & & & \\
,\ x = 20 & & & \\
,\ \mathit{Off} & \mapsto & \dot{x} = -0.1x \wedge x \geq 18 \ [\,] & [\emptyset : x < 19 \gg \tau];\ \mathit{On} \\
,\ \mathit{On} & \mapsto & \dot{x} = 5 - 0.1x \wedge x \leq 22 \ [\,] & [\emptyset : x > 21 \gg \tau];\ \mathit{Off} \\
\mid\ \mathit{Off} & & & \\
\rangle\ . & & &
\end{array}
$$

4 Translation of Chi to hybrid automata

In literature, many different hybrid automata definitions exist. Some definitions require solutions for the continuous variables to be differential functions, e.g. [8, 9]. Other definitions allow the more general case of piecewise differential functions, e.g. [10]. Most hybrid automata definitions do not define urgent transitions, or they define urgent transitions in a restrictive way, as in [11]. In [12], urgent transitions are defined in a general way, using a predicate that defines the maximum sojourn time in a location, but instead of invariants and flow clauses, evolution functions are used. With respect to the meaning of jump clauses, that define the behavior of the variables in action transitions, differences also occur: where in [8] the variables can in principle perform arbitrary jumps unless restricted by the jump predicate, in [11], variables in principle remain unchanged unless changes are enforced by the jump predicate.

None of these hybrid automata definitions is expressive enough to be used as the target for the translation of hybrid χ. Therefore, the translation uses a target hybrid automata definition, called HA_{u} automata, where the u stands for urgency, that uses features from different hybrid automata definitions. In particular, the definition of the jump predicate in combination with a set of changeable variables is based on [9], the solution concept that allows piecewise differentiable functions is based on [10], and the definition of urgent transitions was inspired by [12]. The syntax of the hybrid automata definition HA_{u} is given in Section 4.1. Section 4.2 defines the syntax of the subset χ_{sub} of the χ language that is translated.

4.1 HA_{u} automata definition

A hybrid automaton $HA_{\mathrm{u}} = (X,\ V,\ \mathrm{init},\ \mathrm{inv},\ \mathrm{flow},\ E,\ \mathrm{source},\ \mathrm{target},\ \mathrm{urgent},\ \mathrm{guard},\ \mathrm{jump},\ \Sigma,\ \mathrm{event})$ consists of the following components:

- A finite set of (real-valued) variables $X = \{x_1, \ldots, x_n\}$, the set $\dot{X} = \{\dot{x}_1, \ldots, \dot{x}_n\}$ which denotes the first derivatives of the variables w.r.t. time, and the set $X' = \{x'_1, \ldots, x'_n\}$ which denotes the primed variables that represent values at the conclusion of a discrete change.

- A finite directed multi-graph (V, E), where V denotes a set of vertices (also referred to as control modes or locations) and E denotes a set of edges (control switches).
- Three vertex labeling functions init, inv, and flow that assign to each location $v \in V$ a predicate for initial conditions, invariants and flow conditions, respectively. The free variables of the initial and invariant predicates are from X. The free variables of the flow predicates are from $X \cup \dot{X}$.
- An edge labeling function jump, that assigns to each edge $e \in E$ a set of variables ($\subseteq X$) which are allowed to change and a jump condition which is a predicate whose free variables are from $X \cup X'$.
- An edge labeling function guard, that assigns to each edge $e \in E$ a guard which is a predicate whose free variables are from X.
- An edge labeling function urgent $\in E \to \{\text{true}, \text{false}\}$, that assigns to each edge a boolean: true for an urgent edge, and false for a none-urgent edge.
- A finite set Σ of events, and an edge labeling function event $\in E \to \Sigma$ that assigns to each edge an event.

Usually, an edge e is represented as $e = (v, v')$, which identifies a source location $v \in V$ and a target location $v' \in V$. This representation cannot be used in case of multi-edges (multiple edges with the same source location and target location). To deal with these, two additional functions are defined: function source $\in E \to V$ returns the source location of a given edge, and function target $\in E \to V$ returns the target location of a given edge.

4.2 The χ_{sub} language

The subset χ_{sub} of the χ language that is translated consists of processes $\langle p, \sigma, (\text{dom}(\sigma) \setminus \{\mathsf{time}\}, J, \emptyset, H, \emptyset)\rangle$, where $p \in P_{\text{sub}}$ consists of the guarded atomic process terms: guarded action predicate $b \to W : r \gg l_{\text{a}}$, guarded send $b \to h\,!!\,\mathbf{e}_n$, guarded receive $b \to h\,??\,\mathbf{x}_n$, delay predicate u, consistent deadlock process term δ, and guarded inconsistent process term $b \to \bot$, the unary operators the any delay $[\,]$, repetition $*$, encapsulation $\partial_A(\,)$, urgent communication $\upsilon_H(\,)$ and jump enabling ι_{J+}, and the binary operators sequential composition ; , alternative composition $[\!]$, and parallel composition $\|$. Formally, P_{sub} is defined by:

$$\begin{array}{rcl} P_{\text{sub}} ::= & u \mid \delta \mid b \to \bot \mid b \to W : r \gg l_{\text{a}} \\ \mid & b \to h\,!!\,\mathbf{e}_n \mid b \to h\,??\,\mathbf{x}_n \mid [P_{\text{sub}}] \\ \mid & *P_{\text{sub}} \mid \iota_{J+}(P_{\text{sub}}) \mid P_{\text{sub}} \, ; \, P_{\text{sub}} \\ \mid & P_{\text{sub}} \, [\!] \, P_{\text{sub}} \mid P_{\text{sub}} \, \| \, P_{\text{sub}} \mid \partial_A(P_{\text{sub}}) \\ \mid & \upsilon_H(P_{\text{sub}}) \end{array}$$

In χ_{sub} processes, there are no discrete variables ($\text{dom}(\sigma) = C \cup \{\mathsf{time}\}$), no algebraic variables ($L = \emptyset$), and no recursion variables ($R = \emptyset$).

In χ, the guard operator can be applied to arbitrary process terms. Since it is not possible to translate the guard operator in a general way, the process terms to which the guard operator can be applied are restricted to the consistent deadlock, the action predicate, undelayable send and undelayable receive process terms.

4.3 Translation

In [1], we define a formal translation from χ_{sub} to HA_{u} automata. It is proved that any transition of a χ model can be mimicked by a transition in the corresponding hybrid automaton model and vice versa. This indicates that the translation is correct. Since a manual translation is very time consuming and error-prone, the translation has been automated by implementing it using the programming language Python [13].

5 Verification of χ_{sub} specifications using PHAVer

PHAVer (Polyhedral Hybrid Automaton Verifyer) [14] is a tool for analyzing linear hybrid I/O-automata. If we restrict the linear hybrid I/O-automata to the class of linear hybrid I/O-automata without input variables and without output variables, then this class of linear hybrid I/O-automata is a subclass of the HA_{u} automata as defined in Section 4.1. As a consequence, the χ_{sub} specifications that can be verified using PHAVer are restricted to those specifications that result in HA_{u} automata with linear invariant and flow conditions, and linear jump conditions, where a linear constraint is defined as a constraint over a set of variables X that is of the form $\sum_i a_i v_i + b \lozenge 0$, with $a_i, b \in \mathbb{Z}$, $v_i \in X$, and $\lozenge \in \{<, \leq, =\}$. Furthermore, the χ_{sub} specifications are restricted to those specifications that result in HA_{u} automata which do not contain urgent transitions. This restriction is because in the semantics of the HA_{u} definition, transitions can be urgent, while in the linear hybrid I/O-automata, transitions cannot be urgent.

Note that in [1], the relation between linear hybrid I/O-automata and HA_{u} automata has been formalized.

6 Case study: the water-level monitor

The verification of a χ_{sub} specification using PHAVer is illustrated by means of an example: the water-level monitor, which is taken from [15]. First, the water-level monitor is modeled using χ_{sub}. Then we translate the χ_{sub} specification to a hybrid automaton HA_{u}. Since the obtained hybrid automaton HA_{u} is a linear hybrid I/O-automaton, it is possible to verify properties of this automaton model using PHAVer.

The water-level in a tank is controlled through a monitor, which continuously senses the water-level and turns a pump on and off. When the pump is off, the water-level is denoted by the variable y, drops by 2 per second; when the pump is on, the water-level rises by 1 per second. There is a time delay of 2 seconds between the time that the monitor signals to change the status of the pump and time that the change becomes effective (this is modeled by the variable x). Initially the water-level is 1 and the pump is turned on. The water-level monitor is modeled in χ_{sub} as follows:

$$
\begin{array}{l}
\langle \ \mathsf{cont}\ x, y \\
,\ x = 0, y = 1 \\
\mid\ \dot{x} = 1 \\
\parallel *(\ (\ \dot{y} = 1 \quad \wedge\ y \leq 10\ [\!]\ [y \geq 10 \rightarrow \{x\} : x = 0 \gg \tau]\ ) \\
\quad ;\ (\ \dot{y} = 1 \quad \wedge\ x \leq 2 \ \ [\!]\ [x \geq 2 \rightarrow \{\emptyset\} : \mathrm{true} \gg \tau]\ ) \\
\quad ;\ (\ \dot{y} = -2 \wedge y \geq 5 \ \ [\!]\ [y \leq 5 \rightarrow \{x\} : x = 0 \gg \tau]\ ) \\
\quad ;\ (\ \dot{y} = -2 \wedge x \leq 2 \ \ [\!]\ [x \geq 2 \rightarrow \{\emptyset\} : \mathrm{true} \gg \tau]\ ) \\
\quad) \\
\rangle
\end{array}
$$

This specification is translated into a hybrid automaton HA_{u}, which is shown in Figure 2.

The input language of PHAVer is a straightforward textual representation of linear hybrid I/O-automata [16]. Using a code-generator, the code which can be used as input for PHAVer is automatically generated from the linear hybrid I/O-automaton model.

The safety property that the water-level has to be kept between $1 < y < 12$ has been verified using PHAVer. PHAVer reported that this safety property holds in all locations. In [1], we have a theorem that states that this safety property also holds in the hybrid automaton HA_{u}. Since we proved that any transition of a χ_{sub} specification can be mimicked by a transition in the corresponding hybrid automaton HA_{u} and vice versa, it can be concluded that this safety property also holds in the original χ_{sub} specification.

7 Conclusions and future work

This paper presents the main aspects of the current status of the χ formalism. It illustrates that the χ formalism is closely related to other formalisms from DC and CS.

We describe the translation of a subset of χ to hybrid automata that enables verification of χ specifications using existing hybrid automata based verification tools. Moreover, we discuss the relation between hybrid automata and the linear hybrid I/O automata that are used for the verification tool PHAVer (Polyhedral Hybrid Automaton Verifyer). As a case study, we translate a χ specification of a water-level monitor to a corresponding linear hybrid I/O automaton, and use PHAVer to verify properties.

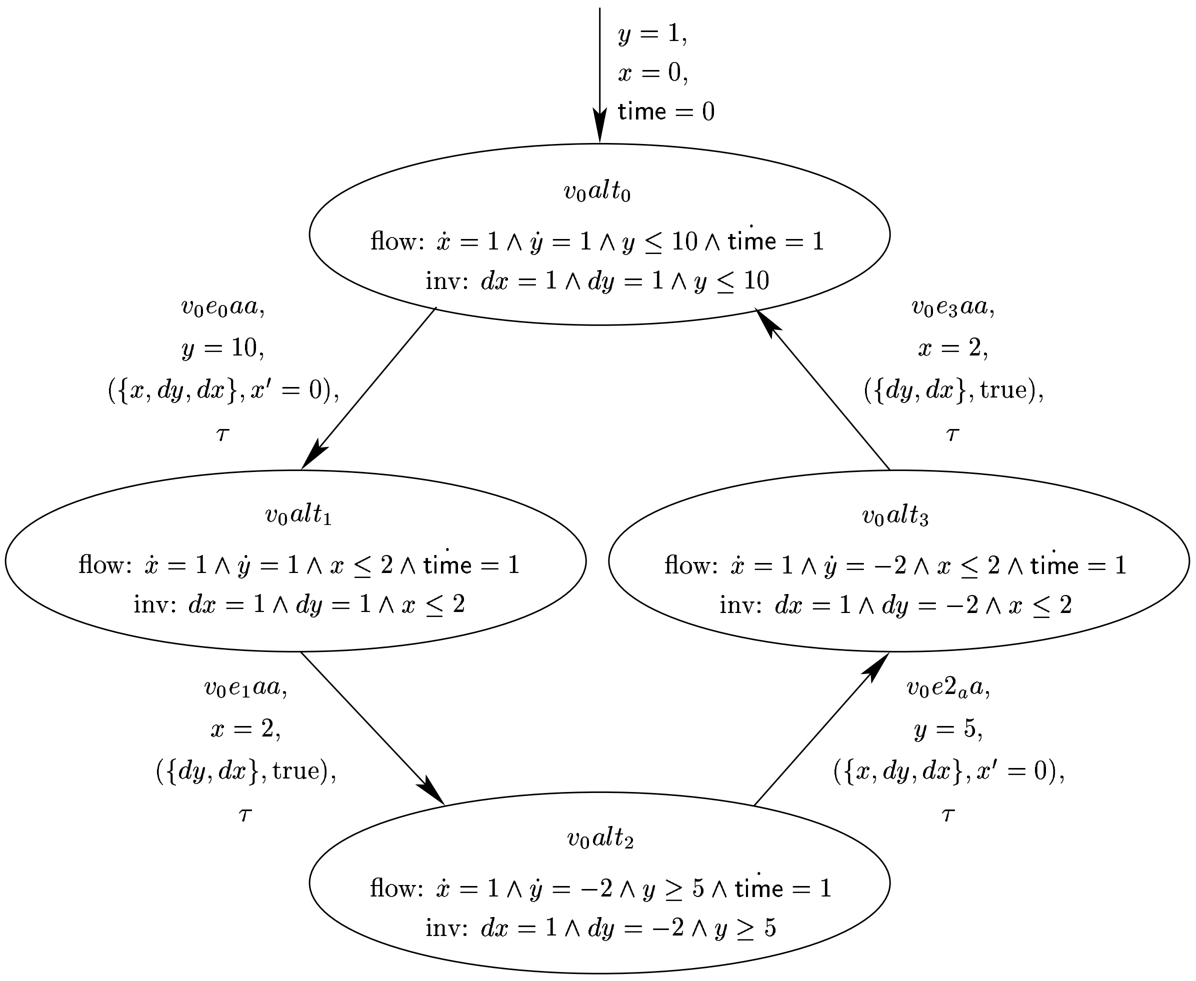

$y = 1$,
$x = 0$,
$\mathsf{time} = 0$
v_0alt_0
flow: $\dot{x} = 1 \wedge \dot{y} = 1 \wedge y \leq 10 \wedge \dot{\mathsf{time}} = 1$
inv: $dx = 1 \wedge dy = 1 \wedge y \leq 10$
v_0e_0aa,
$y = 10$,
$(\{x, dy, dx\}, x' = 0)$,
τ
v_0e_3aa,
$x = 2$,
$(\{dy, dx\}, \text{true})$,
τ
v_0alt_1
flow: $\dot{x} = 1 \wedge \dot{y} = 1 \wedge x \leq 2 \wedge \dot{\mathsf{time}} = 1$
inv: $dx = 1 \wedge dy = 1 \wedge x \leq 2$
v_0alt_3
flow: $\dot{x} = 1 \wedge \dot{y} = -2 \wedge x \leq 2 \wedge \dot{\mathsf{time}} = 1$
inv: $dx = 1 \wedge dy = -2 \wedge x \leq 2$
v_0e_1aa,
$x = 2$,
$(\{dy, dx\}, \text{true})$,
τ
v_0e2_aa,
$y = 5$,
$(\{x, dy, dx\}, x' = 0)$,
τ
v_0alt_2
flow: $\dot{x} = 1 \wedge \dot{y} = -2 \wedge y \geq 5 \wedge \dot{\mathsf{time}} = 1$
inv: $dx = 1 \wedge dy = -2 \wedge y \geq 5$

Figure 2: Generated water-level monitor automaton.

As an alternative to analyze χ specifications using hybrid automata based verification tools, χ simulators can be used to simulate χ specifications. Recently, a symbolic simulator has been developed for χ.

Like in ACP [17] and HyPA [18], a set of basic terms (in a subset of χ) has been defined into which all closed terms (of the subset of χ) can be rewritten using χ properties. This is so-called elimination, which is a useful step for algebraic analysis, because it reduces the complexity of specifications (without recursion variables) by transforming them into simpler forms. The elimination result allows to eliminate the parallel composition from many χ specifications, and it can be regarded as a preprocessing step for the linearization (transformation of a recursive specification into linear form) of χ processes.

Acknowledgments

The authors would like to thank J.C.M. Baeten, D.A. van Beek, M.A. Reniers and J.E. Rooda for the supervision of our Ph.D research work. Furthermore, we thank Rolf Theunissen for performing the case study.

References

[1] Man, K.L., Schiffelers, R.R.H.: Formal Specification and Analysis of Hybrid Systems. PhD thesis, Eindhoven University of Technology (To be accepted)

[2] Naumoski, G., Alberts, W.: A Discrete-Event Simulator for Systems Engineering. PhD thesis, Eindhoven University of Technology (1998)

[3] van Beek, D.A., van den Ham, A., Rooda, J.E.: Modelling and control of process industry batch production systems. In: 15th Triennial World Congress of the International Federation of Automatic Control, Barcelona (2002) CD-ROM.

[4] Fábián, G.: A Language and Simulator for Hybrid Systems. PhD thesis, Eindhoven University of Technology (1999)

[5] van Beek, D.A., Rooda, J.E.: Languages and applications in hybrid modelling and simulation: Positioning of Chi. Control Engineering Practice **8** (2000) 81–91

[6] Bos, V., Kleijn, J.J.T.: Formal Specification and Analysis of Industrial Systems. PhD thesis, Eindhoven University of Technology (2002)

[7] Bos, V., Kleijn, J.J.T.: Automatic verification of a manufacturing system. Robotics and Computer Integrated Manufacturing **17** (2000) 185–198

[8] Henzinger, T.A.: The theory of hybrid automata. In Inan, M., Kurshan, R., eds.: Verification of Digital and Hybrid Systems. Volume 170 of NATO

ASI Series F: Computer and Systems Science. Springer-Verlag, New York (2000) 265–292

[9] Alur, R., Henzinger, T.A., Ho, P.H.: Automatic symbolic verification of embedded systems. IEEE Transactions on Software Engineering **22** (1996) 181–201

[10] van der Schaft, A.J., Schumacher, J.M.: An Introduction to Hybrid Dynamical Systems. Volume 251 of Springer Lecture Notes in Control and Information Sciences. Springer (2000)

[11] Henzinger, T.A., Ho, P.H., Wong-Toi, H.: A user guide to HyTech. In: First International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS. Lecture Notes in Computer Science 1019, Springer Verlag (1995) 41–71

[12] Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: An approach to the description and analysis of hybrid systems. In: Workshop on Theory of Hybrid Systems. (1992) 149–178

[13] Python website: (2005) http://www.python.org.

[14] Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past HyTech. In Morari, M., Thiele, L., eds.: Hybrid Systems: Computation and Control, 8th International Workshop. Volume 3414 of Lecture Notes in Computer Science. Springer-Verlag (2005) 258–273

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

[16] G. Frehse: Language Overview v.0.2.2.1 for PHAVer v.0.2.2, www.cs.ru.nl/ goranf. (2004)

[17] Baeten, J.C.M., Weijland, W.P.: Process Algebra. Volume 18 of Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, United Kingdom (1990)

[18] Cuijpers, P.J.L., Reniers, M.A.: Hybrid process algebra. Journal of Logic and Algebraic Programming **62** (2005) 191–245

An Automated Test Generation Process from UML Models to TTCN-3

Jens R. Calamé
Centrum voor Wiskunde en Informatica, Amsterdam; jens.calame@cwi.nl

Abstract

Conformance testing is a rigorous approach to validate, whether an implementation matches its specification. For this validation, test cases have to be developed from the specification of the system under test (SUT). In this paper, an automated process is provided, which generates TTCN-3 tests from a UML specification. This process makes use of formal methods and the test generator TGV which is based on the enumeration of the state-space of an SUT. Enumerating the state-space of a data-oriented system is a notable challenge, since input and output data from large or infinite domains leads to a state-space explosion.

In the area of model-checking, the approach of data abstraction has been developed to control this problem. In this work, the approach is applied to test generation. A specification is first abstracted with respect to its input and output data, to mitigate the problem of state space explosion due to action parameters. Since information about concrete data is missing now, conditions can become nondeterministic. Thus, the generated test case contains all the relevant traces from the original system as well as additional, spurious traces, introduced by nondeterminism. With constraint-solving, we search for possible test data and automatically filter out these spurious traces.

The general question, followed in my research, is how to support a fully automated test generation process with the means of formal methods. This work has up to now mainly been done in the scope of the ITEA project *TT-Medal* (Test and Testing Methodologies for Advanced Languages, see www.tt-medal.org) in cooperation with a number of European partners like Fraunhofer FOKUS, DaimlerChrysler and Nokia. Project results have been published as project deliverables. Furthermore, results of the work for *TT-Medal* have been published as a technical report [3] and as a conference paper [4].

In this paper, I will first give an overview of the test generation process, as it is planned to result from my research. Then, the actual state of my research is worked out in greater detail and compared to related works. Finally, I give an outlook on the further ideas, which still have to be worked out.

1 Main Ideas

The test generation process is depicted in figure 1. It ideally begins from a system specification in the Unified Modeling Language (UML) and ends in the execution of test cases in the test implementation language TTCN-3 (Testing and Test Control Notation, version 3). In a first step, the UML models for static and dynamic aspects of a system specification, i.e. the definition of datatypes and of system behavior, are transformed to a formal specification in μCRL. This formal model is then transformed into an abstracted one, which is an over-approximation of the original specification. From this abstract specification, abstract test cases are generated by the

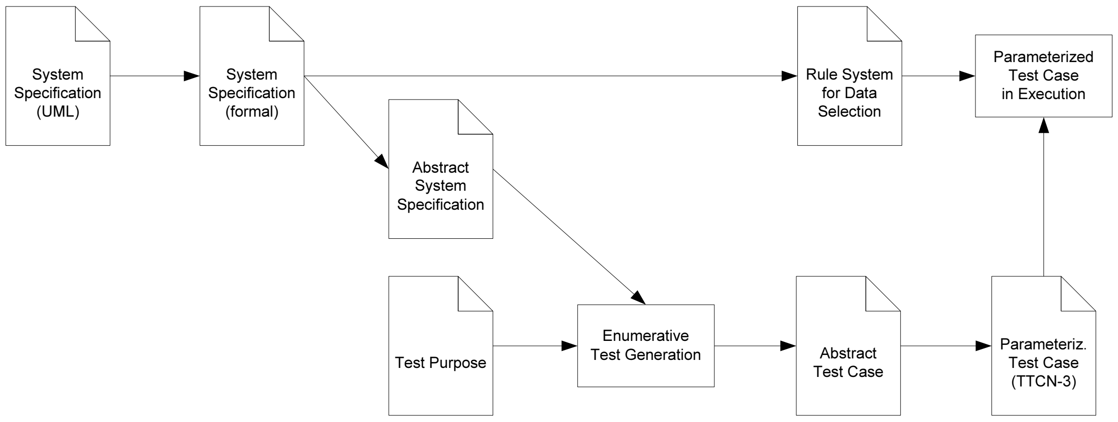

System Specification (UML)
System Specification (formal)
Abstract System Specification
Rule System for Data Selection
Parameterized Test Case in Execution
Test Purpose
Enumerative Test Generation
Abstract Test Case
Parameteriz. Test Case (TTCN-3)

Figure 1: The test generation process

test generator TGV. This generation process is guided by a test purpose. The abstract test cases are available as input-output labeled transition systems (IOLTSs) in Aldébaran and have to be transformed to parameterizable test cases in TTCN-3 before test execution. Still, the test cases are based on the over-approximated specification, so they can contain behavior which is impossible due to the original system specification. To filter out this behavior and to determine useful test data, a rule system has been generated in parallel to the actual test generation, which can be queried before or during test execution.

All steps in the test process from the specification to the generation of the rule system and the test cases in TTCN-3 are fully automated by tools. Only the system specification and the test purpose have to be provided by the software engineer as input for the tools. While test execution also happens automatically, test data selection based on the rule system and test case parameterization are still manual work. It is planned to automatize this step in future. The same holds for the transformation of UML models to a formal specification.

2 Actual Status

Up to this moment, the generation of tests from a formal μCRL specification has been developed (see `http://www.cwi.nl/ calame/dataabstr.html`). In μCRL [2], the system under test (SUT) is described as a linear process equation. This process specifies a set of transitions and for each transition a condition, an action label and the next state of the system. The actions can carry data parameters, which can be variable. From this specification, an IOLTS must be generated to serve as an input for the test generator. This tool, TGV [9], then examines the state space of the SUT to generate test cases. Since the examination of the whole state space is very costly, TGV is guided by a so-called *test purpose*. This is a sketch, which limits the search for test cases to certain aspects of the SUT's behavior, e.g. the main risks of the system.

The difficulty is that the state space of a system cannot simply be generated if its specification allows variables of a large or infinite, for instance numerical, domain. The result would be an infinite or at least very large state space. For this reason, we had to employ data abstraction [4, 8]. Doing so, input and output variables are reduced to a *chaotic* value $\mathbb{T}$. By replacing any of these variables by $\mathbb{T}$, we reduce the resulting state space to a finite one, while at the same time, we do not imply anything about the interaction between the SUT and its environment. $\mathbb{T}$ is introduced for each of the existing datatype D by defining a new datatype $D^{\mathbb{T}}$ which contains $\mathbb{T}_D$ and a lifting function $\kappa : D^{\top\!\!\top} \to D$ for all original values from D. Then, all parameters appearing in the specification are re-typed with the new chaotic datatypes. While constant values stay the same, being lifted with κ, variables are replaced by the $\top\!\!\top$-value according to their type. This leads to a three value logic in the guards, which can also become *chaotic* now. This is solved by introducing another function $may : Bool^{\top\!\!\top} \to Bool$, which returns *true* not only for $\kappa(true)$, but also for $\top\!\!\top_{Bool}$. This leads to a safe over-approximation of the original system, in which more behavior is possible than in the original system.

Test cases are generated from this over-approximation. Since the test cases may also contain behavior, that is originally not possible, we have to take care, that no false test verdicts are assigned during test excution. Furthermore, we have to find test data, which is applicable to execute the test. We achieve this by using constraint-solving. Therefore, the original system specification is, transition by transition, transformed into rules of a Prolog system. Each rule is defined as a 3-ary function with the name of the transition's action. Its parameters are the system's actual state, its state after the transition and a possible set of action parameters. These three parameters are defined as structures, so that the actual number of state variables in the system or of action parameters does not interfere with the rule's arity. The body of the rule is defined by the transition's guard, which must be satisfied to execute the action in the SUT. Having defined these rules, a query is generated for each test case. Such a query can be defined in a way that it provides all relevant data parameters (input as well as output parameters), which are needed to instantiate and execute the test case under consideration. In its body, this rule conjuncts all action invocations as they are defined in one trace of the test case to either a *pass* verdict (the implementation conforms its specification) or an *inconc* verdict (specification and tets case do not match). For more than one trace, one can define separate queries. If there is a solution for a trace, values for the data parameters are returned. Using these values, the test case can be instantiated and executed.

The previous paragraph described the static use of queries on the rule system. We have, in theory, also developed an algorithm for dynamic querying. In principle, this algorithm works as follows: One trace of the test case is pre-solved statically, then the test case is instantiated and executed. If now at some point, the SUT does not react in the predicted way, this does not necessarily mean a fault, but can also be allowed nondeterministic behavior. In this case, the test system dynamically tries to find another trace to a *pass* verdict in the test case under consideration of the SUT's last reaction. If such a trace can be found, the test case is re-instantiated and executed further. If only a trace to an *inconc* verdict can be found instead, it unveils a mismatch between the test case and the SUT's specification. In this case, the *inconc* verdict is assigned to the test execution and the test stops. If no matching trace can be found at all, the test ends with a *fail* verdict. In this case, the implementation does not match its specification.

In addition to the above-mentioned work on data abstraction and constraint solving for test generation, some work has already been done on the transformation of parameterizable test cases to TTCN-3. Therefore, a mapping of μCRL datatypes to TTCN-3 datatypes and of abstract test cases as IOLTSs to TTCN-3 test cases has been made, but is still ongoing work.

We have evaluated our approach to test generation on the Common Electronic Purse Specifications (CEPS) [5]. They define a protocol for electronic payment using a multi-currency smart-card. Such a card has a number of slots, each corresponding to one currency and a balance. The functionality of this card, like loading and paying, is defined in the CEPS. We have written a specification for the inquiry and load functionality of CEPS in μCRL and applied our approach of data abstraction on it. This was necessary, since the card functionality strongly depends on datatypes with a numerical domain. We have then generated the rule system as well as test cases for the *load transaction* of the card. The state space generation for CEPS and the state space reduction took 16 minutes 5 seconds on a cluster of five 2.2GHz AMD Athlon 64 bit single CPU computers with 1 GB RAM each (operating system: SuSE Linux 9.3, kernel 2.6.11.4-20a-default). Using TGV, we generated two test cases without loops: one of 594 states with 597 transitions and another one of 109 states with 111 transitions. Test case generation took 0.65 seconds and 0.42 seconds, respectively, on a workstation with one 2.2GHz AMD Athlon XP 32 bit CPU and 1 GB main memory (operating system: Redhat Linux Fedora Core 1, kernel 2.4.22-1.2199.nptl).

3 Future Work

Several theoretical as well as practical aspects of this work require further consideration and research. On the practical side, they are mainly related to the UML- and TTCN-3 integration of the test generation approach. The dynamic constraint solving approach has yet been worked out only theoretically, so that a practical realization and experiments are necessary here. Furthermore, TTCN-3 test cases are already generated, but they cannot yet be executed since an implementation under test or at least an appropriate simulation of one is missing. It is planned to execute the test cases in parallel to a simulation of the SUT's specification. Therefore, an execution environment for μCRL specifications must be provided which can be connected to a TTCN-3 test system. Furthermore, data selection and test case parameterization are planned to be at least tool-supported, but ideally fully automatized. The last practical aspect, which has to be examined, is the use of UML models of the SUT instead of formal models. In order to do so, UML must be restricted and formalized like it has been done in the projects *Agedis* [1] and *Omega* [11]. Then it is planned to provide an automatized transformation from UML to μCRL.

On the theoretical level, several questions have not yet been answered. The correct handling of internal τ-steps of an SUT is, for instance, not completely resolved. We are working on the level of black-box testing, so that internal τ-steps do not appear in the test cases. However, they must be part of the query on the rule system, since they also drive the evolution of the SUT's state during execution. Furthermore, the effect of data abstraction on suspension traces in the SUT has not yet been investigated. Finally, it must be proven, that the presented approach generates a complete set of test cases. This means that if an SUT does not conform its specification there is a test which can find this failure.

4 Related Work

The closest to our approach is *symbolic test generation* [7, 10, 12]. This method works directly on higher-level specifications given as input-output symbolic transition systems (IOSTSs) without enumerating their state space. Given a test purpose and a specification, their product is built. The coreachability analysis is in these cases over-approximated by Abstract Interpretation [6].

The purpose and usage of abstraction techniques in our approach is conceptually different from the one of symbolic test generation, since we use a data abstraction that mitigates infinity of external data. This enables us to use existing enumerative test generation techniques for the derivation of abstract test cases which are then instantiated with concrete data derived by constraint solving. In the symbolic test generation approach, approximate coreachability analysis is used to prune paths potentially not leading to *pass*-verdicts. Both approaches are valid for any abstraction leading to an over-approximation of the SUT's behavior. They both also employ constraint solving to choose a single testing strategy during test execution, so that more case studies are needed to conclude which approach is more suitable for which class of systems.

References

[1] Automated Generation and Execution of Test Suites for DIstributed Component-based Software (AGEDIS). http://www.agedis.de.

[2] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. van Langeveld, B. Lisser, and J.C. van de Pol. μcrl: a toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)*, pages 250–254. Springer-Verlag, 2001.

[3] Jens R. Calamé. Specification-based test generation with tgv. Technical Report SEN-R0508, Centrum voor Wiskunde en Informatica, May 2005. ISSN 1386-369X.

[4] Jens R. Calamé, Natalia Ioustinova, Jaco van de Pol, and Natalia Sidorova. Data abstraction and constraint solving for conformance testing. In *Proc. of the 12th Asia Pacific Software Engineering Conference (APSEC 2005).* IEEE Computer Society, To appear 2005.

[5] CEPSCO. *Common Electronic Purse Specifications, Technical Specification*, May 2000. Version 2.2.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM Press.

[7] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test generation based on symbolic specifications. In J. Grabowski and B. Nielsen, editors, *FATES 2004*, number 3395 in LNCS, pages 1–15. Springer-Verlag, 2005.

[8] N. Ioustinova, N. Sidorova, and M. Steffen. Synchronous closing and flow abstraction for model checking timed systems. In *Proc. of the Second Int. Symposium on Formal Methods for Components and Objects (FMCO'03)*, volume 3188 of *LNCS*. Springer, 2004.

[9] C. Jard and T. Jéron. TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004.

[10] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva. Symbolic test selection based on approximate analysis. In *11th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)Volume 3440 of LNCS*, Edinburgh (Scottland), April 2005.

[11] Correct Development of Real-Time Embedded Systems (Omega). http://www-omega.imag.fr.

[12] Eléna Zinovieva-Leroux. *Méthodes symboliques pour la génération de tests de systèmes reactifs comportant des données.* PhD thesis, Université de Rennes, 2004.

Analysis and Implementations of MSC Specifications

Carsten Kern

Lehrstuhl für Informatik II, RWTH Aachen, Germany
`kern@informatik.rwth-aachen.de`

Abstract. In this paper we consider the formal model of (*compositional*) *Message Sequence Graphs* – (C)MSGs for short – which provide a standardized modeling language for specifying communication protocols. We are mainly interested in detecting classes of implementable protocols and therefore define properties on (C)MSGs that ensure their implementability. The implementation of a CMSG in our context means to deploy the model of communicating finite-state machines for constructing automata which recognize (up to synchronization messages) the same language as a given (C)MSG. These syntactic properties can be checked by our tool MSCAN. To our knowledge there is no other tool that provides a protocol designer with a likewise great variety of facilities to analyze (C)MSGs. The complexity of algorithms for testing the membership of a CMSG to a certain property class ranges from PTIME to CO-NP completeness. Because of this discrepancy in complexity we group the property classes into two inclusion hierarchies. The first one is a property hierarchy and the second one is a language hierarchy which identifies equivalent property classes. Using the results of these diagrams eases the verification of protocol characteristics. For the final phase of system design we also introduce an implementation of the very important class of *local-choice* CMSGs, exhibiting the nice property of deadlock freedom.

1 Introduction

The complexity of today's software systems is increasing rapidly and so is the need for the employment of formal methods to guarantee their reliability. It is desirable to apply formal methods already in the early stages of system design to avoid costly and extensive reimplementation and redesign. When developing communicating systems, it is a widespread design practice to start with drawing scenarios showing the intended interaction of the system in mind. *Message Sequence Charts* (MSCs), a modeling language at a high level of abstraction, provide a prominent notion to further this approach. They are widely used in industry, standardized [ITU98,ITU99], and similar to UML's sequence diagrams [Ara98]. An MSC depicts a single partially ordered execution sequence of a system. Moreover, it defines a collection of processes, which, in its visual representation, are drawn as vertical lines and interpreted as time axes. An arrow from one line to a second corresponds to the communication events of sending and receiving a message. But the MSC standard does not only allow to specify single scenarios. To make MSCs a flexible specification language, it also supports *choice*, *concatenation*, and *iteration*, which gives rise to *Message Sequence Graphs* (MSGs). To ease a protocol designer's life our goal was, on the one hand, to provide an analysis tool [MSC05] offering a great variety of properties that may be checked for given CMSGs to ensure important protocol characteristics. In chapter 2 the most important of these properties will be described. On the other hand we built inclusion hierarchies for relating these properties and finding classes of equal expressiveness. These hierarchies will be introduced in chapter 3. As the implementation of a protocol is the final step in a protocol-design cycle we propose an implementation for the very important class of *local-choice* CMSGs in chapter 4 and close with a short section on the tool's web page in chapter 5.

2 CMSGs and their properties

In the following we define the most important objects we are going to deal with throughout this paper, namely compositional Message Sequence Charts and Message Sequence Graphs.

Definition 1 (Compositional Message Sequence Chart). *A compositional Message Sequence Chart M consists of:*

- *a finite, non-empty set of processes $\mathcal{P}$,*
- *a finite, non-empty set of events $E = \biguplus_{p \in \mathcal{P}} E_p = S \uplus R$, occurring on the processes and being divided into send events (S) and receive events (R),*
- *a function t labeling the events*
  $t : E \to Act \quad , \quad t(e) := \begin{cases} p!q \ , \text{ if } e \in E_p \cap S & (\textit{process } p \textit{ sends a message to } q) \\ p?q \ , \text{ if } e \in E_p \cap R & (\textit{process } p \textit{ receives a message from } q) \end{cases}$
- *a partial and injective function $m : S \dashrightarrow R$ matching send events to receive events of the correct type (**Note:** not every send event needs to have a corresponding receive event and vice versa), and*
- *a partial order $< \subseteq E \times E$ on the events.*

This definition can easily be extended to message contents but for the sake of brevity we omitted them here. We call a CMSC an MSC if m is total and bijective.

CMSCs usually describe a single execution of a system. If we want to specify the system itself we need (compositional) Message Sequence Graphs.

Definition 2 (Compositional Message Sequence Graph). *A Compositional Message Sequence Graph G consists of:*

- *a graph $\langle V, R \rangle \quad (V \neq \emptyset \ , \ R \subseteq V \times V)$,*
- *a non-empty set of start nodes $V^0 \subseteq V$, a set of end nodes $V^f \subseteq V$, and*
- *a function λ that assigns a CMSC to each node of the graph.*

If, in the CMSG definition, λ maps to the set of MSCs than we call it an MSG.

But using MSGs for protocol design easily leads to problems. There is, for example, no possibility to create an MSG for the famous *alternating-bit protocol* [Tan03]. Thus we will use CMSGs throughout this paper to avoid such problems.
The following definition will be used in the subsequent subsection for describing global communication behavior of CMSGs.

Definition 3 (Communication Graph (cf. [Gen05])). *The communication graph of a CMSC $M = \langle \mathcal{P}, E, \mathcal{C}, t, m, < \rangle$ is defined as the digraph containing a node for each active process (i.e., a process with at least one event) in M and there is an edge from node p to node q whenever there is at least one send event on process p with type $p!q$ and one receive event on process q with type $q?p$.*

2.1 CMSG-Properties

If we design systems, their underlying protocols usually have to fulfill properties like, for example, "*sending a message always expects an acknowledgement of the receiver*" also known as the *regularity*-property. In this section we want to itemize further examples of important characteristics of protocols. We will describe most of them intuitively because formal definitions would be too involved.

- **general case:** The general case has no restrictions regarding communication behavior.
- **globally cooperative:** A CMSG G is called *globally cooperative* if every CMSC labeling a cycle in G has got a connected communication graph. This property assures that on these cycles every process participating in the communication interacts with all others from time to time. This requirement seems to be essential for allowing synchronization between the autonomous processes.

- **regular:** This property restricts the cycles of G to have strongly connected communication graphs and thus ensures that for each sent message the receipt of an acknowledgement becomes possible.
- **locally cooperative:** This property changes the focus from globally observable behavior to local characteristics checking nodes and pairs of nodes instead of strongly connected graph components.
- **local-choice:** This property assures that in choice nodes, i.e., in nodes which possess more than one direct successor, a group of processes (*weak* local-choice) or even a single process (*strong* local-choice) may choose the next node to enter. After the choice has been made this information is passed to the other processes. The local-choice property will play an important role at implementing CMSGs in chapter 4.
- **local:** The *local* property checks the local-choice property not only for each branching node but for all nodes contained in the graph. As with local-choice we distinguish between the weak and the strong version of locality.

The weak versions of local and local-choice, of course, induce some kind of non-determinism into the protocol because different processes may select different successor nodes for system continuation.

In figure 2, on page 5, we present an overview over the complexity for checking the properties we just introduced.

3 Property and Language Hierarchy

3.1 Property Hierarchy

Due to the huge complexity gap between the property classes concerned with local characteristics of CMSGs and the ones describing global communication structures, we introduce a property hierarchy which depicts the inclusion relation of our syntactic properties. A diagram like this eases the analysis of systems in the way that if *easy-to-check* (according to figure 2) properties are disproved for a certain CMSG the designer would not have to check for properties higher in the inclusion hierarchy or vice versa if a *harder-to-check* property would be verified for the given CMSG all properties in the inclusion relation lying beneath it would automatically hold.
On the left side of figure 1 we see the property hierarchy for the properties from chapter 2.1.

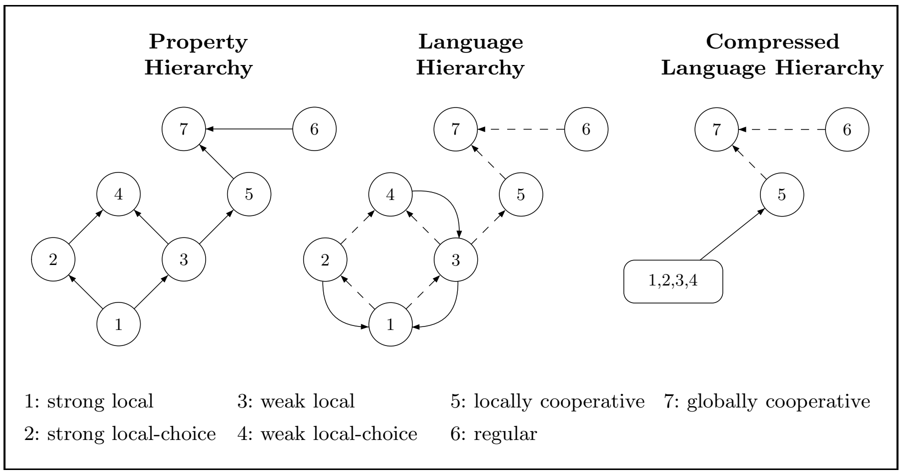


Fig. 1. CMSG-property and language hierarchy

Theorem 1 (cf. [Ker05]). *The previously introduced properties obey the strict (,i.e., "$\subsetneq$") inclusion hierarchy depicted on the left side of figure 1.*

3.2 Language Hierarchy

Now we want to detect classes of properties which can be proved to be equivalent. For that purpose we define the language a CMSG is describing.

Definition 4. *The language $\mathcal{L}(G)$ described by a CMSG G is the set of all MSCs representing the accepting paths of G.* (A path in a CMSG G is called *accepting* if it starts in a start node of G and ends in one of its end nodes.)

Moreover two CMSGs G and G' are said to be *equivalent* if they describe the same language, i.e., if $\mathcal{L}(G) = \mathcal{L}(G')$. Now we can specify the language classes we have proved to be equivalent.

Definition 5. *Language classes:*

$$\begin{array}{lll} \theta - loc\mathfrak{CMSG} := \{\mathcal{L}(G) \,|\, G \text{ is a } \theta \text{ local CMSG}\} & \theta \in \{weak, strong\} \\ \theta - lc\mathfrak{CMSG} := \{\mathcal{L}(G) \,|\, G \text{ is a } \theta \text{ local-choice CMSG}\} & \theta \in \{weak, strong\} \end{array}$$

In the following we present the results we have achieved so far describing the center and the right part of figure 1.

Theorem 2 ([Ker05]). *Every weak, respectively strong local-choice CMSG $G = \langle V, R, V^0, V^f, \lambda\rangle$ can be transformed into an equivalent weak, respectively strong local CMSG G' of size $\mathcal{O}(|V|)$.*

Then it directly follows from theorem 2 and figure 1:

Corollary 1 ([Ker05]).

- $strong - lc\mathfrak{CMSG} = strong - loc\mathfrak{CMSG}$
- $weak - lc\mathfrak{CMSG} = weak - loc\mathfrak{CMSG}$

A second transformation relates weak and strong locality.

Theorem 3 ([Ker05]). *Every weak local CMSG $G = \langle V, R, V^0, V^f, \lambda\rangle$ can be transformed into a strong local CMSG of size $\mathcal{O}(|V|^2)$.*

From theorem 3 and figure 1 it directly follows:

Corollary 2 ([Ker05]).

- $weak - loc\mathfrak{CMSG} = strong - loc\mathfrak{CMSG}$

Using the transformations from theorems 2 and 3 we are able to construct strong local out of weak local choice CMSGs and hence to eliminate the non-determinism resulting from the *weak* versions of the properties, namely *weak* locality and local-choice. This result is important if one wants to implement CMSGs. If a given CMSG fulfills the weak local-choice property without satisfying the strong version, then there is at least one branching node on which two distinct processes may decide on the further progress of the system. If these processes make different choices the implementation may run into a deadlock. Thus it is desirable to eliminate such behavior whenever this is possible. Our transformation algorithms resolve the problem by transforming the weak property versions into the strong ones. In the next section we will concentrate on an implementation for local-choice CMSGs keeping the results from this section in mind.

4 CMSG implementations

The main goal of defining the properties from section 2.1 is to identify classes of implementable (C)MSGs. Implementability in this context means that the CMSG can be transformed into communicating automata which represent the processes in the CMSCs exchanging messages among each other. For this purpose we deploy the formal model of *communicating finite-state machines* (CFMs).

Definition 6 (Communicating Finite-State Machines). *A Communicating Finite-State Machine over a process set $\mathcal{P}$ is defined as: $\mathcal{A} = \langle (\mathcal{A}_p)_{p\in\mathcal{P}}, Sync, F\rangle$.*

- *For every $p \in \mathcal{P}$, the CFM $\mathcal{A}$ possesses a finite automaton $\mathcal{A}_p = \langle S_p, s_p, \longrightarrow_p\rangle$ over the finite, non-empty set of actions Act_p with:*
  - *a finite, non-empty set S_p of local states,*
  - *a starting state $s_p \in S_p$ and*
  - *a transition relation $\longrightarrow_p \subseteq S_p \times Act_p \times Sync \times S_p$ describing the state changes of $\mathcal{A}_p$ (Sync is a finite set of synchronization messages for synchronizing the local automata. It is basically needed for deadlock prevention.)*
- *and a set of global final states $F \subseteq \prod_{p\in\mathcal{P}} S_p$.*

Note 1. The local automata communicate with each other over error-free fifo channels while the CFM changes from a *configuration* to another by letting one automaton perform a write or read to or from one of its channels. The CFM begins in a starting configuration where all local automata are situated in their start state and all buffers are empty and ends in a final configuration, i.e., a state from F is reached and all buffers are empty again. We call a configuration of a CFM a *deadlock* if there is no possibility to reach a final configuration.

Having an autonomous automaton for each process results in a problem. Without synchronization, the automata do not know about the other automatas' choices and thus may run into different branches of the CMSG resulting in a deadlock of the system. Hence, for implementing CMSGs, we have to use the set of synchronization messages *Sync*.
In general deadlocks result from the discrepancy between the global control of the CMSG switching from node to node and the local autonomous processes not being aware of the graph's choice. But if we require the local-choice property a CMSG becomes implementable without deadlocks ([Ker05]).

Theorem 4 ([Ker05]). *Let G be a strong local-choice CMSG, then G is implementable without deadlocks.*

Combining this result and the results from figure 1 yields the following:

Corollary 3 ([Ker05]). *Every (weak and strong) local and local-choice CMSG G is implementable without deadlock.*

Finally we want to present an overview of the complexity classes and some implementability results from [GMSZ02], [GKM04] and [Ker05]. We divided the implementability property into *normal* and *deadlock-free* implementability.

PROPERTY	COMPLEXITY	IMPLEMENTABILITY/ DEADLOCK FREEDOM
local	**polynomial time**	**YES/YES**
local-choice	**polynomial time**	**YES/YES**
locally coop.	**polynomial time**	**YES/ ?**
regular	**Co-NP complete**	**YES/NO**
globally coop.	**Co-NP complete**	**YES/NO**
general case	**$\geq$ CO-NP complete**	**NO/NO**

Fig. 2. Complexity classes an implementability results (cf. [GMSZ02], [GKM04], [Ker05])

5 Additional Information and Acknowledgement

As we already mentioned, in addition to our theoretical results, we implemented a tool named MSCan which provides possibilities for specifying and analyzing CMSGs. For further information on this project please consult the tool's web page:

`http://www-i2.informatik.rwth-aachen.de/MSCan`

I am deeply grateful to Benedikt Bollig for supervising this work.

6 Future Work

As part of my future work, we plan to conduct research on quantitative extensions of Message Sequence Charts (MSC) and Message Sequence Graphs (MSG). We intend to focus in particular on Life Sequence Charts (LSC, cf. figure 3). One of the drawbacks of MSCs and MSGs is that these formalisms do not provide ample means to distinguish between what “must” happen and what “may” happen in a communication systems’ behaviour. This shortcoming hampers the specification of scenarios for realistic system design. LSC diminish these problems by allowing to distinguish between *mandatory*, *optional*, and *illegal* or *forbidden* behavior which *must*, *may* and *must not* occur, respectively.

The quantitative extensions of Message Sequence Charts include modeling events that appear with a certain probability distribution as well as unreliable channels where messages can get lost with a given likelihood. LSCs do only exhibit possibilities for describing system behavior qualitatively but in this way will be extended to cope with quantitative system aspects. These extensions need not be restricted to the LSC body but might also be integrated into the LSCs head where the preconditions of the chart are constituted. These prerequisites have to be fulfilled to execute the LSC body. We also anticipate an extension of MSGs where we model the different transition probabilities for changing from one node to another, especially in branching nodes.

These extensions need to be formally defined, and must be equipped with a rigid semantics. To that purpose it is planned to investigate extensions of partial-order models (e.g., event structures and pomsets) as well as tree- and trace-based models. Like in the reported work in this paper, elementary properties of such quantitative LSCs/MSCs will be defined and classified. The relation with existing probabilistic extension of statecharts will be studied. The ultimate goal is to come to define a framework in which probabilistic scenarios can be used to synthesize system components in a semi-automated manner.

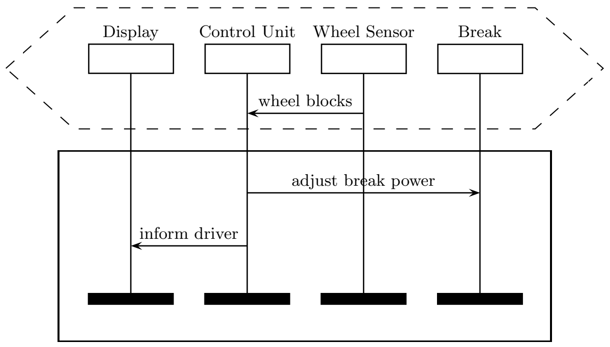


Fig. 3. small LSC for the antiblock system

References

[Ara98] João Araújo. Formalizing sequence diagrams. In *Proceedings of the OOPSLA’98 Workshop on Formalizing UML. Why? How?*, 1998.

[Gen05] B. Genest. Compositional Message Sequence Charts (CMSCs) Are Better to Implement Than MSCs. In *TACAS*, pages 429–444, 2005.

[GKM04] B. Genest, D. Kuske, and A. Muscholl. A Kleene theorem for a class of communicating automata with effective algorithms. unpublished manuscript, 2004.

[GMSZ02] B. Genest, A. Muscholl, H. Seidl, and M. Zeitoun. Infinite-state high-level MSCs: Model-checking and realizability. In *Proceedings of ICALP 2002*, volume 2380 of *Lecture Notes in Computer Science*. Springer, 2002.

[ITU98] ITU-TS Recommendation Z.120 Annex B: Formal Semantics of Message Sequence Charts, 1998.

[ITU99] ITU-TS Recommendation Z.120: Message Sequence Chart 1999 (MSC99), 1999.

[Ker05] C. Kern. MSCan – Ein Tool zur Analyse von Message Sequence Charts. Master's thesis, Dept. of Computer Science, RWTH Aachen, Germany, August 2005.

[MSC05] MSCan. *MSCan – Message Sequence Chart analysis tool web page*, 2005. `http://www-i2.informatik.rwth-aachen.de/MSCan/`.

[Tan03] A. S. Tanenbaum. *Computer Networks.* Prentice-Hall International, fourth edition, 2003.

Frameworks based on templates for rigorous model-driven development

Nuno Amálio

Dept. of Computer Science, University of York, YO10 5DD, UK
namalio@cs.york.ac.uk

November 7, 2005

Abstract

The engineering of systems that are acceptably correct is a hard problem. On the one hand, semi-formal modelling approaches that are used in practical, large-scale system development, such as the UML, are not amenable to formal analysis and consistency checking. On the other hand, formal modelling and analysis requires a level of competence and expertise that is not common in commercial development communities, and formal approaches are not well integrated with the rest of the development process. The thesis presented here proposes an approach to build engineering environments (or frameworks) for rigorous model-driven development (MDD) that combines semi-formal notations with formal modelling languages. The approach proposes a formal template language to capture patterns of formal development and enabling an approach to formal proof with templates (meta-proof). This allows the development of catalogues of patterns (represented as templates) and meta-theorems for frameworks. The thesis develops a framework for the combined use of UML and the formal language Z.

1 Introduction

Model-driven development (MDD) is an approach that elects models, rather than code, as primary artifacts of software development, using them for description and analysis of software systems. Models can be built to describe succinctly the domain and required behaviour of a system, being useful in the stages of construction and maintenance of software. The analysis of models uncovers flaws and brings up fundamental issues related with the requirements and design of the system. It is in the early stages of development, when code does not yet exist, that model analysis is most rewarding, exposing problems that cost much more if not discovered until later.

Software engineering practice relies on *semi-formal techniques* for MDD. These techniques are based on diagrammatic-based notations (e.g. UML) to describe different aspects of systems. They are called semi-formal because the notations have a formal syntax, but no formal semantics. Semi-formal notations have a graphical nature and this makes them appealing: they are intuitive, providing good sketches of different aspects of systems that are easily assimilable. Moreover, semi-formal notations, such as UML and entity-relationship diagrams, are defacto modelling idioms among software engineers. However, these notations lack a formal semantics and this constitutes a serious limitation: models expressed in these notations are likely to be ambiguous, inconsistent and not amenable to mechanical semantic analysis, and tool support becomes limited to superficial syntactic analysis and transformations. The problem with the semantics is aggravated by the fact that these notations have, in fact, many semantic interpretations: the choice of semantics becomes a matter of convenience, developers use one semantics or the other depending on the kind of problem at hand.

Formal techniques, on the other hand, are substantially less used in practice. Formal modelling languages (e.g. Z, B, CSP, Alloy) are based on mathematics and formal logic and they embody years of research and best practice in formal development. The models resulting from formal languages are precise, unambiguous, and amenable to formal analysis. However, formal modelling and analysis requires a level of competence and expertise that is not common in commercial development communities, and formal approaches are not well integrated with the rest of the development process.

There have been numerous attempts to introduce formal techniques in mainstream modelling practice, by using them combined with semi-formal notations. This work is a step in the right direction, but is has just touched the surface, much more is yet to be done. Some approaches define a semantics for the semi-formal notations in a mathematical formalism, making the resulting models unambiguous, but they don't explore the semantics for analysis; sometimes the chosen formalism is not the most suitable for analysis, or the semantics is too complex, compromising the extension of the work. Other approaches propose a translation from diagrams into a formal modelling language, thus giving a precise semantics to the diagrams, and making available the analysis mechanism of the formal language, but giving no support whatsoever to the analysis itself; to explore analysis one is required to be an expert in the use of the formal language. Other approaches spend effort in developing yet another special-purpose formal language, rather than building on existing mature work. Most approaches don't take into account the multiple-semantics of semi-formal notations (e.g.UML), defining one single semantics; however, developers may want alternative semantics for certain high-level modelling concepts, even within the same development; if one is to cope with this feature a flexible and practical approach to define semantics of semi-formal notations is required.

The doctoral thesis will propose an approach to build environments (or frameworks) to support MDD and that are designed to address the needs of one specific problem domain [1]. The approach aims at engineering systems that are acceptably correct, and so to allow engineers to build, analyse and refine models of software systems in a *sound but practical* way. The approach proposes frameworks that combine formal and semi-formal methods with the diagrams acting like a graphical interface for the formality that lies underneath. The aim is to hide the formality as much as possible, but this is not always possible; sometimes one expert in the formal technique is required, but non-experts can still participate and engage in model-development by drawing diagrams. By designing frameworks targeted to specific problem domains, the multi-semantic interpretation problem of semi-formal notations is addressed: in one framework the diagrams have one semantics, the one suitable to models problems of a certain domain. The approach puts a strong emphasis on *patterns* as a key to enhance the practicality of frameworks. To represent patterns of formal development, the thesis proposes the formal template language (FTL), which enables an approach to proof with template representations [5]. This allows not only the representation of patterns of formal development (e.g. a model structure), but also to reason about these representations using meta-proof (e.g. calculating a pre-condition or proving an initialisation theorem).

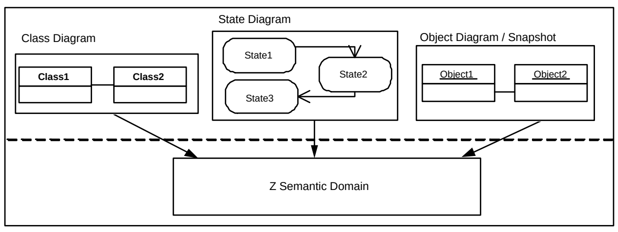


Figure 1: Models in the $UML + Z$ framework.

To evaluate the approach, the thesis develops one framework for the modelling general sequential systems following the object-oriented paradigm. This framework uses the formal specification language Z and the UML, and so it is called $UML + Z$ (figure 1).

In the rest of this paper we discuss FTL and meta-proof. Then we discuss the modelling, analysis and refinement components of the $UML + Z$ framework.

2 FTL and Meta-proof

Often, computer scientists find the need to represent sentences of some language that have a particular form and that cannot be represented in the language itself. Templates are representations of these generic sentences, which, upon instantiation, yield specific language sentences.

The thesis presents a formal language to express templates, FTL [5]. The language is simple: essentially all it does is text substitution. It is also general in the sense that it can be applied to any formal language; so far it has been used with Z.

For example, we can use FTL to capture the state of a Z promoted ADT [11]:

$$\lessdot P \gtrdot == [\ \lessdot ids \gtrdot : \mathbb{P} \lessdot ID \gtrdot ;\ \lessdot st \gtrdot : \lessdot ID \gtrdot \pfun \lessdot S \gtrdot \mid \mathrm{dom} \lessdot st \gtrdot = \lessdot ids \gtrdot\ ]$$

This introduces five parameters, P, ids, ID, st and S, which are to be replaced by some text values when instantiated. This template can be instantiated to yield:

$$Bank == [\ accounts : \mathbb{P}\ ACCID;\ accountSt : ACCID \pfun Account \mid \mathrm{dom}\, accountSt = accounts\ ]$$

Any formal sentence or sentences of some formal language (here Z) can be represented as templates expressed in FTL. Is it possible to reason (or do proof) with these template representations? If it would be possible, that would have substantial practical value. It would mean that *reuse* could be brought to the level of proofs: *meta-theorems* for certain templates would be proved once, but could be applicable every time those templates are instantiated. This approach of proof with templates is called *meta-proof.* First, the practical value of meta-proof is motivated with an example.

In Z, the introduction of a description of the state space of an abstract data type (ADT), such as *Bank* above, into a specification, entails a demonstration that the description is consistent: at least one state satisfying the description does exist. This involves defining the initial state of the ADT (the so-called initialisation) and proving that the initial state does exist (the initialisation theorem). The initialisation of *Bank* assumes that in the initial state there are no accounts:

$$BankInit == [\ Bank\ ' \mid accounts' = \varnothing \land accountSt' = \varnothing\ ]$$

To demonstrate the consistency of *Bank*, one is required to discharge the conjecture $\vdash ?\ \exists\ BankInit \bullet true$, which is automatically discharged in the Z/Eves theorem prover [9].

This proved theorem applies to the *Bank* ADT only. The question is: does it apply to all promoted ADTs that are similar in form to *Bank*? And if it does, can this result be proved once and for all, so that developers don't have to do it again and again?

The empty initialisation of a promoted ADT and the associated conjecture is represented with templates :

$$\lessdot P \gtrdot Init == [\ \lessdot P \gtrdot\ ' \mid \lessdot ids \gtrdot' = \varnothing \land \lessdot st \gtrdot' = \varnothing\ ]$$
$$\vdash ?\ \exists \lessdot P \gtrdot Init \bullet \mathrm{true}$$

This *meta-conjecture* is true for all *well-formed* instantiations of these templates, that is, instantiations that result in type-correct Z sentences. The argument is as follows. For all well-formed instantiations, P holds a name; id and st hold names of variables of type set and these two names must be distinct; and ID and S must be names referring to sets. If we expand the template schemas above using the laws of the schema calculus, and apply the one-point rule, we get the formula

$$\vdash ?\ dom\ \varnothing = \varnothing \land \varnothing \in \lessdot ID \gtrdot \land \varnothing \in \lessdot ID \gtrdot \pfun \lessdot S \gtrdot$$

which is true for any well-formed instantiation of ID and S. We have just established a *meta-theorem*: the initialisation conjecture of all promoted ADT with empty initialisation that are instantiations of these templates is *true*. This gives us a nice property of *true by construction*:

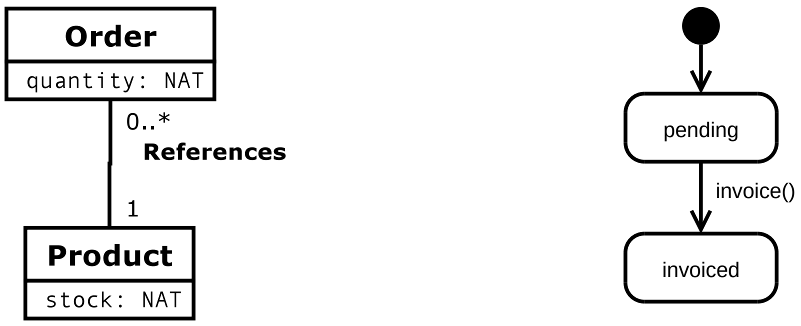


Figure 2: The UML class diagram of the trivial ordering system and the UML state diagram for the class *Order*.

whenever these templates are instantiated to build promoted ADTs, their initialisations are *true* by appeal to this meta-theorem.

The argument outlined in this meta-proof is rigorous and valid, but it is not formal. To follow a formal approach towards meta-proof, a formal semantics has been given to the language [5]. This allows the definition of proof rules for Z template expressions, which are proved by appeal to the semantics of the language [5].

3 Modelling

The modelling component of frameworks is defined following denotational semantics [10]. A catalogue of templates captures the structure of the *semantic domain*; every formal sentence that makes the framework's models is generated by instantiating a template from the catalogue. A set of diagram types constitutes the *syntactic domain*; every diagram of the framework's models is an instance of these diagram types. Finally, there is the *semantic mapping*, which maps diagrams into templates instantiations.

For the $UML + Z$ framework, the thesis builds a Z semantic domain for object-oriented (OO) models [2], which expresses typical OO structures, such as class, association, and systems, the concept of inheritance (or class specialisation), and the UML idioms of composition and association class. The thesis will also present the modelling catalogue of $UML + Z$, which includes templates that capture the sentences of the Z semantics domain and meta-theorems. The $UML{+}Z$ framework is illustrated for a simple example. The semantic mapping of $UML + Z$ is left for future work.

Figure 3 presents the class diagram of a simple ordering system and the state diagram of the class *Order*. The Z representation of these diagrams is obtained by instantiating templates of the $UML + Z$ catalogue. For example, to represent the class *Order*, we need to generate its Z intensional definitions (types, state space, initialisation, operations and finalisation, see [2] for details) by instantiating templates with information coming from the diagrams. The Z type representing the possible values of the state diagram for *Order* is obtained as follows:

$$\lessdot\, Cl \,\gtrdot ST ::= \lessdot\, initSt \,\gtrdot [\![\;\mid \lessdot\, oSt \,\gtrdot \;]\!] \qquad OrderST ::= pending \mid invoiced$$

The state space of *Order* is defined by instantiating the template of state intension for classes with a state diagram:[1]

$$\begin{array}{|l}
\lessdot\, Cl \,\gtrdot \\
\hline
state : \lessdot\, Cl \,\gtrdot St \\
[\![\lessdot\, at \,\gtrdot : \lessdot\, atT \,\gtrdot]\!] \\
\hline
\lessdot\, ICL \,\gtrdot \\
\hline
\end{array}
\qquad
\begin{array}{|l}
Order \\
\hline
state : OrderSt \\
quantity : \mathbb{N} \\
\hline
\mathrm{true} \\
\hline
\end{array}$$

[1]Note that the attribute *quantity* comes from the class diagram and *state* from the state diagram.

The initialisation is also defined by instantiating the initialisation template for such classes:

$$\begin{array}{|l}
\hline
\lessdot\ Cl \gtrdot Init \\
\lessdot\ Cl \gtrdot{}' \\
[\![\lessdot\ in \gtrdot ? : \lessdot\ inT \gtrdot]\!] \\
\hline
state' = \lessdot\ initSt \gtrdot \\
[\![\lessdot\ at \gtrdot{}' = \lessdot\ ival \gtrdot]\!] \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
OrderInit \\
Order' \\
quantity? : \mathbb{N} \\
\hline
state' = pending \\
quantity' = quantity? \\
\hline
\end{array}$$

To consistency of *Order* is checked by using the meta-theorems of the *UML* + *Z* catalogue. The initialisation conjecture of classes such as *Order* is simplified to:

$$\begin{array}{l}
\vdash ?\ \ \exists\ [\![\lessdot\ in \gtrdot ? : \lessdot\ inT \gtrdot]\!] \bullet \\
\qquad \lessdot\ ICL \gtrdot [\ [\![\lessdot\ at \gtrdot := \lessdot\ ival \gtrdot]\!]\ ]\ [\![\ \ \land \lessdot\ ival \gtrdot \in \lessdot\ atT \gtrdot]\!]
\end{array}$$

The instantiation of this template for *Order* gives:[2]

$$\vdash ?\ \ \exists\ quantity? : \mathbb{N} \bullet quantity? \in \mathbb{N}$$

And this is trivially true: the state of *Order* is consistent. Note that in this case all we get from meta-proof is a simplification, the nice property of *true by construction* cannot be obtained.

The full specification of the trivial ordering system is given in [3].

4 Analysis

The analysis component defines an approach to analyse the framework's models. The thesis will propose an approach to analyse *UML* + *Z* models based on formal proof and Catalysis [6] snapshots [4]: *snapshot-based* validation.

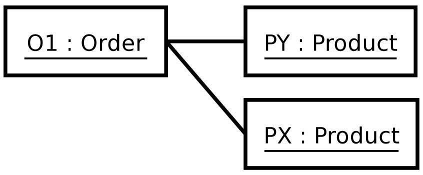


Figure 3: An Order associated with two products.

Snapshots are object diagrams that describe states of the modelled system and these states can be represented in the Z OO semantic domain as an instance of the Z model. The approach uses snapshots and formal proof to test the model: we prove that there is a state of the modelled system satisfying the snapshot (an existence proof). Snapshots can be used to describe one single state or a state transition, and they can describe either states or transitions that should be accepted by the system (positive testing) or those that should not be accepted by the system (negative testing).

In the trivial Bank System, the customer has a very specific requirement: each order must reference only one product. To test this requirement we can draw a snapshot describing a system state that should not be accepted by the model of the system (figure 3): an order referring to two products. This snapshot is represented in Z as:

$$\begin{array}{|l}
\hline
StSnap1 \\
System \\
\hline
orders = \{oO1\} \land orderSt = \{oO1 \mapsto O1\} \\
products = \{oPX, oPY\} \land productSt = \{oPX \mapsto PX, oPY \mapsto PY\} \\
references = \{oO1 \mapsto oPX, oO1 \mapsto oPY\} \\
\hline
\end{array}$$

(Here, *System* is the Z schema that defines the system as whole, see [2] for details.)

This state should not be accepted by the system, thus we prove the conjecture (the negation of the positive case):

[2]The conjecture to prove without the use of meta-theorems would be, $\exists\ OrderInit \bullet true$.

$\vdash ?\ \neg (\exists\ StSnap1 \bullet \text{true})$

And this conjecture is true (we have proved it in Z/Eves [9]), meaning that the state described by the snapshot is not accepted by the model of the system.

5 Refinement

The refinement component defines a strategy to refine models built using the framework, and includes a catalogue of model transformations (refactorings).

The thesis will present a strategy to refine $UML + Z$ models, based on the theory of refinement for Z, and some example model transformations. The idea is to explore FTL and meta-proof to capture refactorings and to substantially reduce the proof overhead associated with these transformations. The process is similar to the one followed in modelling: (a) refactorings and associated correctness conjectures are captured with templates; (b) meta-proof is applied upon these representations to simplify (and in some cases fully prove) those correctness conjectures. This will allow model transformations to be carried our by instantiating templates: the associated correctness proofs can then be simplified to smaller proofs after applying the associated meta-theorems. The refinement component of $UML + Z$ is still under development.

6 Related Work

The work presented here borrows ideas from several sources. The idea of frameworks that are customised to problem domains is inspired in the work of Michael Jackson [7, 8], who advocates that different problems demand different methods and the use of different concepts and notations.

Catalysis [6], the modelling method based on the UML, has many features that are analogous to the approach presented here. The method presents the ideas of model frameworks and templates to make models reusable assets, model system states by using object-diagrams (snapshots), model refinement with UML diagrams, and the idea of defining semantics of UML constructs adapted to the context in which they are used. However, in Catalysis this is done informally. The approach presented here tries to take similar ideas into development, but within a formal framework. Moreover, the template notation used in Catalysis has less features than FTL (only parameters; FTL has choice and lists), and it is not defined formally.

7 Conclusions

The thesis outlined here proposes an approach to build frameworks for rigorous MDD. These frameworks combine semi-formal and formal modelling languages. Associated with this approach, is a formal template language (FTL) enabling meta-proof, which allows the representation of patterns of formal modelling and proof as templates. FTL allows the factoring of model-level definitions to the meta-level and formal proof with template representations (meta-proof) to establish meta-theorems that are proved once and used every time the associated templates are instantiated.

The approach presented here offers the following benefits:

- FTL and meta-proof helps to make formal methods more practical. FTL allows the factoring of patterns of formal modelling and proof to the meta-level, so that the same effort can be reused. In $UML + Z$, all Z is generated by template instantiation, so that the specifier doesn't have to come up with a proper specification structure again and again; this process can also be made automatable. The developer's effort in terms of proof can also be reduced; conjectures expressing certain desired properties are simplified and in some cases fully proved, so that the developer either has a simpler verification task or doesn't have any task at all.

- The emphasis on template patterns and frameworks tailored to problem domains foster knowledge reuse. MDD frameworks encapsulate valuable modelling knowledge and experience about the domain in the form of patterns (captured with templates); this grows as the framework is applied to more problems. The same body of work can be reused and adapted to meet the needs of other problems (either within the same framework, or for new frameworks exploring new problem domains).

The aim is to make the formal language hidden to the user as much as possible. In *UML* + *Z* this could not be fully achieved. At least one expert is required to write Z operation specifications and invariants that are not expressible in terms of UML diagrams. Nevertheless, the *UML* + *Z* framework allows non-Z experts to engage in the modelling and analysis effort, by drawing class, state and object diagrams, and yet offering the benefits of formal development.

References

[1] N. Amálio. *Frameworks based on templates for rigorous model-driven develpment.* PhD thesis, Department of Computer Science, University of York, 2006. to appear.

[2] N. Amálio, F. Polack, and S. Stepney. An object-oriented structuring for Z based on views. In H. Treharne et al., editors, *ZB 2005: International Conference of B and Z users*, volume 3455 of *LNCS*, pages 262–278. Springer, 2005.

[3] N. Amálio, F. Polack, and S. Stepney. *Software Specification Methods: an overview using a case study*, chapter *UML* + *Z*: UML augmented with Z. Hermes Science, 2006. to appear.

[4] N. Amálio, S. Stepney, and F. Polack. Formal proof from UML models. In J. Davies et al., editors, *ICFEM 2004: Int. Conference on Formal Engineering Methods*, volume 3308 of *LNCS*, pages 418–433. Springer, 2004.

[5] N. Amálio, S. Stepney, and F. Polack. A formal template language enabling meta-proof. submitted, 2005.

[6] D. D'Sousa and A. C. Wills. *Object Components and Frameworks with UML: the Catalysis approach.* Addison-Wesley, 1998.

[7] M. Jackson. Formal methods and traditional engineering. *The Journal of Systems and Software*, 40(3):191–194, Mar. 1998.

[8] M. Jackson. *Problem Frames: Analyzing and structuring software development problems.* Addison-Wesley, 2001.

[9] M. Saaltink. The Z/EVES system. In *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*. Springer, 1997.

[10] R. D. Tennent. The denotational semantics of programming languages. *Commun. ACM*, 19(8):437–453, 1976.

[11] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* International series in computer science. Prentice-Hall, 1996.

Acknowledgements. This work was funded by the Portuguese Foundation for Science and Technology, grant 6904/2001. The PhD was supervised by Prof. Susan Stepney and Dr. Fiona Polack, both with the department of Computer Science of the University of York. Some of the work presented here was supervised by Prof. Augusto Sampaio, during Amálio's academic visit to the Federal University of Pernambuco, Brazil, which was was funded by the industrial sponsors of the department of computer science of the University of York. All funding and supervision is gratefully acknowledged.

A passive Dolev-Yao intruder that reads xor

Mohammad Torabi Dashti
CWI, Amsterdam

Abstract

We extend the passive Dolev-Yao intruder with a decision algorithm to check if a (secret) term can be derived with arbitrary use of xor from a set of observed communicated messages. The extended intruder is formally specified in the process algebraic language μCRL and subsequently used to find a (known) flaw in an implementation of Shamir's three pass protocol.

1 Introduction

The de facto standard of intruder model, i.e. the Dolev-Yao intruder after [DY83], is based on black-box cryptography assumption. There, it is assumed that all the cryptographic primitives used in security protocols are absolutely trustful and the intruder cannot learn more by looking through their detailed implementation. This assumption, however, is not the most realistic. There are several well-known attacks on security protocols that exploit algebraic properties of the underlying cryptosystems. In this paper, we extend the passive Dolev-Yao intruder with the ability to comprehend and exploit the algebraic properties of the xor operator, which is widely used in security protocols.

We motivate this work by briefly describing a known attack on Shamir's three pass protocol [Mas92]. This protocol aims at sharing a secret key over a seamless communication channel, hence only passive eavesdropping is possible for the intruder. The protocol assumes that the participants have access to a commutative encryption algorithm, namely:

$$E_y(E_x(M)) = E_x(E_y(M))$$

It immediately implies that $D_x(E_y(E_x(M))) = E_y(M)$. The protocol is as the following, where A wants to share a secret key K with B.

1. $A \rightarrow B : E_{K_A}(K)$
2. $B \rightarrow A : E_{K_B}(E_{K_A}(K))$
3. $A \rightarrow B : E_{K_B}(K)$

Here K_A and K_B are private keys of A and B, respectively. For composing m_3 (message 3 in the protocol), A simply uses the commutative property of the encryption algorithm. As an implementation of this protocol, the xor operator can be used as the encryption function, i.e. $E_x(M) = x \oplus M$, which is apparently commutative. Such an implementation, however, is flawed as the intruder can derive K (and so K_A and K_B) by computing $m_1 \oplus m_2 \oplus m_3$. Clearly an intruder based on the black-box cryptography assumption is not able to find such an attack.

2 A Decision Algorithm for xor

A passive intruder has an increasing set of knowledge, gathered from all passed messages over the communication media. To equip such an intruder with the power of comprehending xor expressions, we need to provide it with an algorithm to decide whether a term can be derived from a set of terms with arbitrary applications of xor. Assume that X and Y are sets of symbolic terms, like $a \oplus b$. We define

$$\begin{aligned}
&X = \{x_1, \cdots, x_n\}\\
&Y = \{y_1, \cdots, y_m\}\\
&X \oplus Y \overset{def}{=} \{x_i \oplus y_i | i \in \{1, \cdots, n\}, j \in \{1, \cdots, m\}\}\\
&X^0 = \{0\},\ \forall n > 0.\ X^n \overset{def}{=} X \oplus X^{n-1}
\end{aligned}$$

When $|X|$ represents the cardinality of the finite set X, we observe that

$$\forall i > 0.\ X^i \subseteq \cup_{j \in \{1, \cdots, |X|\}} X^j.$$

So what can be derived from a set X of terms with arbitrary applications of xor is contained in the set $\cup_{j \in \{1, \cdots, |X|\}} X^j$. Based on this fact, the following algorithm decides whether a term t can be derived from a set X of terms or not:

```
1. $W = X,\ i = 1$.
2. if $t \in W$, then $X \vdash t$; end.
3. $W' = W \cup X^{i++}$
4. if $W' = W$, then $X \nvdash t$; end.
5. $W = W'$; goto 2
```

Note that this algorithm is terminating according to the discussion above. The algorithm has been implemented in μCRL, extending the passive Dolev-Yao intruder, and then successfully used to find the mentioned flaw in Shamir's three pass protocol based on xor encryption.

3 Future Work

We have extended a passive Dolev-Yao intruder with a decision algorithm for xor expressions. Having the same extension for an active intruder is not straightforward, due to the difficulty of considering type flaw attacks. Another future research would be to look for decision procedures for other commonly used algebraic functions in cryptosystems, such as Diffie-Hellman exponentiation.

References

[DY83] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Trans. on Information Theory*, IT-29(2):198–208, 1983.

[Mas92] J. L. Massey. Contemporary cryptology: An introduction. In G. J. Simmons, editor, *Contemporary Cryptology - The Science of Information Integrity*, pages 1–39. IEEE Press, 1992.

Component-Interaction Automata for Specification and Verification of Component Interactions

[Extended Abstract]

Pavlína Vařeková[*] and Barbora Zimmerova[*]
Faculty of Informatics
Masaryk University in Brno
Czech Republic
{xvareko1,zimmerova}@fi.muni.cz

ABSTRACT

The paper presents an automata-based language, *Component-Interaction automata*, designed for specification of component interactions in hierarchical component-based software systems. The language aims to provide a direct and desirable way of modelling component interactions which is meant to be transparent and understandable thanks to the orientation to component-based systems and their specifics. After a brief introduction that identifies the research problems we discuss related work and identify the main issues that motivated us to evolve a new language to support the mentioned purpose. One of the main differences to related specification languages is that Component-Interaction automata use a flexible composition of components that can be parametrized by *architecture description of a system* (hierarchical assembly of primitive components) and other characteristics. Moreover, the model is designed to preserve all important interaction information to provide a rich base for further application of formal methods. As a distinct to related languages (I/O automata, Interface automata, Team automata) it naturally preserves information about the components which participated in a synchronization and about the hierarchical structure of an overall system. After a definition of the Component-Interaction automata language, we also discuss possible applications of a model and domains of future work.

1. INTRODUCTION

With increasing use of component-based and service-oriented principles in practical software development process, new issues demanding application of formal methods in this domain arise. Let us name for instance component compatibility, composability, substitutability and refinement of two components, checking of temporal properties of component interactions or issue of assembly strategies for composing a system satisfying particular properties.

For further investigation of mentioned issues, proper specification of component behaviour and interactions in assembled hierarchical system is indispensable[1]. The right specification language should be able to cover all specifics of component-based systems and to model component interactions without lost of any important information (like which components participated in the synchronization or how does the hierarchy of a modelled system look like) and with respect to the architecture of a system[2].

In the rest of the paper, we examine some of the current specification languages usable for this purpose and after discussion of their reduced applicability to this context we introduce an automata-based language, *Component-Interaction automata*, designed with the primary intention of specification and verification of component interactions in component-based software systems.

2. SPECIFICATION LANGUAGES

Generally, there are several groups of languages that can be used for description of component behaviour and interactions in hierarchical component-based software systems. The two of them that are the closest to the given purpose are *architecture description languages (ADLs)* and formal *automata-based languages*.

2.1 Architecture description languages

Architecture description languages, like *Wright [2]*, *Darwin/Tracta [10, 11]*, and *SOFA [12]*, allow to specify both statical (system architecture, bindings among components) and dynamical (component behaviour, interactions) aspects of hierarchical component-based systems. These languages are very close to practice and often address many practical issues which can arise in real life systems. Additionally, they use to be supported by tools.

As the languages are oriented more to the practical aspects of the architecture design, they often fail to be useful when more formal issues of the model need to be addressed. Sim-

[*]The authors have been supported by the grant No. 1ET400300504

[1]Let us remark that by a component we mean an encapsulated unit interacting with the environment solely through its interfaces (provided/required). The interaction among components is then taken on a client/supplier principle where the client component requires (through a required interface) a service on the supplier component which provides it (through a provided interfaces). The service represents a synchronization point of components' interaction

[2]Architecture of a system represents a hierarchical assembly of a system from primitive components.

ilarly in a domain of verification, these languages usually support verification of only a small fixed set of properties. They are often limited by the underlying behavioural model which is usually designed for one particular type of communication and notion of erroneous behaviour and does not cover some important interaction properties that may arise through the composition.

2.2 Automata-based languages

Some of the automata-based languages suitable for specification of component interactions are *I/O automata [9, 8]*, *Interface automata [5, 6]*, and *Team automata [7, 3]*. These languages are highly formal and general and usually allow for straightforward application of formal methods and verification algorithms.

Unfortunately, when we were examining the usability of these languages for general application of formal methods for investigation of component interaction properties of component-based systems, we detected several issues that reduce the suitability of these languages in this context.

The essential issue is that these languages capture the component behaviour only and do not take into account the architecture of an overall system which also plays an indispensable role in interaction specification. That is restrictive especially when the interconnection structure of a system differs from the complete interconnection space determined by the actions shared among components. For example when we need to express that two components cannot synchronize in the composition, even if they share the same action, because the assembly of a system do not allow them to interact (they are not connected one to each other).

Next thing to mention is that these automata-based languages do not naturally preserve some interaction information important for formal reasoning about the resulting system. That is for instance which primitive components participated in an interaction over a particular action, or how does the hierarchy of a modelled system look like. Additionally, they are often limited to one strict type of communication and notion of erroneous behaviour (I/O automata, Interface automata).

Other issue worth noting is that not every set of components described in these languages can be composed. There are several restriction often defined over a disjointness of sets of input, output and internal actions. That means that we are for example unable to compose several automata when some of them contain the same output action (I/O automata, Interface automata) even if it is quite natural in practical component-based systems, for instance when two components are using the same service of another component.

In some cases, relabelling and transformation of the component automata before each composition would be sufficient to express desired features. But the price we would have to pay for it lies in considerable state expanding, untransparency, and uncomfortable use of the model. Moreover, there are many features (like input enableness at I/O automata, explicit indication of erroneous behaviour at Interface automata, or lost of information about participants of a synchronization at Team automata) which would be non-trivial to overcome.

3. COMPONENT-INTERACTION AUTOMATA LANGUAGE

The mentioned issues motivated us to evolve a verification-oriented automata-based formal model, *Component-Interaction automata [4]*, which combines the benefits of both architecture description languages and automata-based languages and is designed with the primary purpose oriented to component-based systems and their specifics. The language makes it possible to model all interesting aspects of component interactions and to preserve important interaction information during the composition as well. One of the essential differences to the previously discussed languages is that the Component-Interaction automata use a flexible composition of components with respect to the architecture description (component assembly) and other characteristics of a modelled system. The language can therefore accept an ADL description as an input and respect it when composing the components what enables it to adopt most of the benefits of architecture description languages. On the other hand, it is a formal automata-based model allowing for direct application of formal methods.

3.1 Definition of a component-interaction automaton

The language should be defined in a way that it is able to describe systems composed from components capturing the information important for detecting all interaction properties. For capturing the information about interacting components of an automaton we among others need to remember what are the primitive components of the automaton and how they were composed. Therefore it is necessary to have each primitive component in the automaton associated with a unique name. In case of Component-Interaction automata the names are natural numbers.

Information about the composition order that was used to compose a particular component is in Component-Interaction automata preserved within a *hierarchy of component names*. Every hierarchy of component names is an n-tuple (where $n \in \mathbb{N}$) whose items correspond to lower level hierarchies of component names that can be primitive or composed. A primitive hierarchy of component names for a component with a numerical name 1 can take form of (1) or $(((1)))$ for instance, and a composed hierarchy of component names for a composition of components 1 and 2 can take form of $((1),(2))$, $(1,(2))$, or $(1,2)$ for instance.

Definition: *Hierarchy of component names* is

- every tuple $H = (n_1, \ldots, n_m)$, where $m \in \mathbb{N}$, $n_1, n_2, \ldots, n_m \in \mathbb{N}$, and $n_1, n_2, \ldots, n_m$ are pairwise different; a set of component names corresponding to H is $S_H = \{n_1,\, n_2,\, \ldots,\, n_m\}$
- every tuple $H = (H_1, H_2, \ldots, H_m)$, where $m \in \mathbb{N}$, $H_1, H_2, \ldots, H_m$ are hierarchies of component names, and $S_{H_1}, S_{H_2}, \ldots, S_{H_m}$ are pairwise disjoint; a set of component names corresponding to H is $S_H = \bigcup_{i=1}^{m} S_{H_i}$
- nothing else

A set of hierarchies of component names is denoted $\mathcal{H}$.
A hierarchy of component names $H \in \mathcal{H}$ is called *primitive* iff $|S_H| = 1$.

Now we can proceed to a definition of component-interaction automaton that is a labelled transition system (with structured labels) enriched with a hierarchy of component names whose composition the automaton represents.

Definition: A *component-interaction automaton* (or *CI automaton* for short) is a 5-tuple $\mathcal{C} = (Q, Act, \delta, I, H)$ where

- Q is a finite set of *states*,
- Act is a finite set of *actions*, $\Sigma = ((S_H \cup \{-\}) \times Act \times (S_H \cup \{-\})) \setminus (\{-\} \times Act \times \{-\})$ is a set of *labels* called an *alphabet*,
- $\delta \subseteq Q \times \Sigma \times Q$ is a finite set of *labelled transitions*,
- $I \subseteq Q$ is a nonempty set of *initial states* and
- $H \in \mathcal{H}$ is a hierarchy of component names.

The labels are triplets of a form $(-, a, n_2)$, $(n_1, a, -)$, or (n_1, a, n_2) and accordingly are of a type *input*, *output*, or *internal* respectively.

- The input label $(-, a, n_2)$ represents that the component n_2 receives an action a as an input.
- The output label $(n_1, a, -)$ represents that the component n_1 sends an action a as an output.
- The internal label (n_1, a, n_2) represents that the component n_1 sends an action a as an output, and synchronously the component n_2 receives the action a as an input.

Example 3.1: Let us consider a simple example from figure 1 (modelled in UML 2.0) capturing a component System consisting of three subcomponents *Client1*, *Client2* and *Database*. The *Database* component provides an insert service, may log the service internally and returns a confirmation of successful insertion. Both *Clinet1* and *Client2* request an insert service first and wait for the confirmation afterwards.

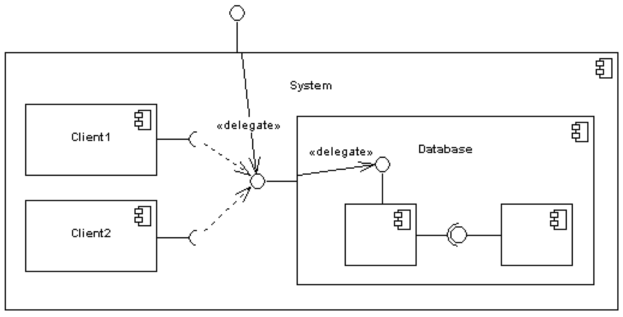


Figure 1: Component model of a simple system

CI automata $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$ for the components *Client1*, *Client2* and *Database*, respectively, are in figure 2. An action i corresponds to inserting some data to the database, an action l corresponds to logging some action and d corresponds to a confirmation that the action was successfully performed (done).

$$\begin{tikzcd} \mathcal{C}_1 : & {} \arrow[r] & 0 \arrow[r, shift left, "{(1,i,-)}"] & 1 \arrow[l, shift left, "{(-,d,1)}"] \end{tikzcd}$$

Hierarchy of component names: (1)

$$\begin{tikzcd} \mathcal{C}_2 : & {} \arrow[r] & 0 \arrow[r, shift left, "{(2,i,-)}"] & 1 \arrow[l, shift left, "{(-,d,2)}"] \end{tikzcd}$$

Hierarchy of component names: (2)

$$\begin{tikzcd} \mathcal{C}_3 : & {} \arrow[r] & 0 \arrow[r, shift left, "{(-,i,3)}"] & 1 \arrow[l, shift left, "{(3,d,-)}"] \arrow[r, shift left, "{(3,l,4)}"] & 2 \arrow[l, shift left, "{(4,d,3)}"] \end{tikzcd}$$

Hierarchy of component names: $(3, 4)$

Figure 2: CI automata $\mathcal{C}_1$, $\mathcal{C}_2$ and $\mathcal{C}_3$

The simplest form of component-interaction automaton according to the hierarchy of component names is an automaton representing one individual component only (with primitive hierarchy of component names).

Definition: A component-interaction automaton $\mathcal{C} = (Q, Act, \delta, I, H)$ is *primitive* iff H is a primitive hierarchy of component names.

Example 3.2: Automata $\mathcal{C}_1$ and $\mathcal{C}_2$ in figure 2 are primitive, automaton $\mathcal{C}_3$ is not.

In some cases it is useful to abstract from an inner hierarchy of a CI automaton and consider it as a primitive one to make the system less complex for further verification.

Definition: Let $\mathcal{C} = (Q, Act, \delta, I, H)$ be a component-interaction automaton. Then a component-interaction automaton $\mathcal{C}' = (Q, Act, \delta', I, (n))$ is *primitive to* the component-interaction automaton $\mathcal{C}$ iff

- $n \in \mathbb{N}$, $n \notin S_H$,
- $(q, (n, a, n), q') \in \delta'$ iff $\exists n_1, n_2 \in \mathbb{N} : (q, (n_1, a, n_2), q') \in \delta$,
- $(q, (-, a, n), q') \in \delta'$ iff $\exists n_2 \in \mathbb{N} : (q, (-, a, n_2), q') \in \delta$,
- $(q, (n, a, -), q') \in \delta'$ iff $\exists n_1 \in \mathbb{N} : (q, (n_1, a, -), q') \in \delta$.

Example 3.3: Considering the automata from figure 2, both the automaton $\mathcal{C}_1$ is primitive to $\mathcal{C}_2$, and the automaton $\mathcal{C}_2$ is primitive to $\mathcal{C}_1$. An example of a CI automaton primitive to the automaton $\mathcal{C}_3$ is in figure 3.

$$\begin{tikzcd} \mathcal{C}_4 : & {} \arrow[r] & 0 \arrow[r, shift left, "{(-,i,5)}"] & 1 \arrow[l, shift left, "{(5,d,-)}"] \arrow[r, shift left, "{(5,l,5)}"] & 2 \arrow[l, shift left, "{(5,d,5)}"] \end{tikzcd}$$

Hierarchy of component names: (5)

Figure 3: CI automaton $\mathcal{C}_4$

3.2 Composition of CI automata

Component-interaction automata can be composed to form a higher level component-interaction automaton. The transition set of the resulting automaton is defined over a complete transition space representing all potentially feasible transitions of the system. The complete transition space for a set of component-interaction automata consists of all transitions capturing that (1) one of the automata follows its original transition or (2) two of the automata synchronize over a complementary transition.

Notation: Let $\mathcal{I} \subseteq \mathbb{N}$ be a finite set with cardinality m, and let for each $i \in \mathcal{I}$, Q_i be a set. Then $\Pi_{i\in\mathcal{I}} Q_i$ denotes the set $\{(q_{i_1}, q_{i_2}, \ldots, q_{i_m}) \mid (\forall j \in \{1,\ldots,m\} : q_{i_j} \in Q_{i_j}) \wedge \{i_1, i_2, \ldots, i_m\} = \mathcal{I} \wedge (\forall j_1, j_2 \in \{1,\ldots,m\} : j_1 < j_2 \Rightarrow i_{j_1} < i_{j_2})\}$. If $\mathcal{I} = \emptyset$ then $\Pi_{i\in\mathcal{I}} Q_i = \emptyset$.
For $j \in \mathcal{I}$, $proj_j$ denotes the function $proj_j : \Pi_{i\in\mathcal{I}} Q_i \to Q_j$ for which $proj_j((q_i)_{i\in\mathcal{I}}) = q_j$.

Definition: Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a system of component-interaction automata such that the sets S_{H_i}, where $i \in \mathcal{I}$, are pairwise disjoint. Then the *complete transition space* for $\mathcal{S}$ is $\Delta_{\mathcal{S}} = \Delta_{\mathcal{S},old} \cup \Delta_{\mathcal{S},new}$, where

$\Delta_{\mathcal{S},old} = \{(q, (o_1, a, o_2), q') \mid q, q' \in \Pi_{i\in\mathcal{I}} Q_i,$
$o_1, o_2 \in \mathbb{N} \cup \{-\},\ \exists j \in \mathcal{I} :$
$((proj_j(q), (o_1, a, o_2), proj_j(q')) \in \delta_j \wedge$
$\wedge\ \forall i \in (\mathcal{I} \setminus \{j\})\ proj_i(q) = proj_i(q'))\}$

$\Delta_{\mathcal{S},new} = \{(q, (n_1, a, n_2), q') \mid q, q' \in \Pi_{i\in\mathcal{I}} Q_i,$
$n_1, n_2 \in \mathbb{N}, \exists j_1, j_2 \in \mathcal{I}, j_1 \neq j_2 :$
$((proj_{j_1}(q), (n_1, a, -), proj_{j_1}(q')) \in \delta_{j_1} \wedge$
$(proj_{j_2}(q), (-, a, n_2), proj_{j_2}(q')) \in \delta_{j_2} \wedge$
$\wedge\ \forall i \in (\mathcal{I} \setminus \{j_1, j_2\})\ proj_i(q) = proj_i(q'))\}$

Example 3.4: The complete transition space for a set of CI automata $\{\mathcal{C}_i\}_{i\in\{1,2,3\}}$ from figure 2 is in figure 4. Every state $(q_1, q_2, q_3) \in Q_1 \times Q_2 \times Q_3$ is represented as a sequence $q_1q_2q_3$ for short.

Now we can proceed to the composition of a set of CI automata which is defined in a flexible manner as a CI automaton over the set, and can take several forms for the same set of CI automata. In particular, the CI automaton over the set is defined as a product automaton whose transition set is a subset of a complete transition space. Thanks to this fact the resulting automaton may consist of only those transition that are really feasible in a system – according to the architectural assembly for instance.

Notation: Let $\mathcal{I} \subseteq \mathbb{N}$ be a finite nonempty set with cardinality m, and $\{H_i\}_{i\in\mathcal{I}}$ be a set. Then $(H_i)_{i\in\mathcal{I}}$ denotes the tuple $(H_{i_1}, H_{i_2}, \ldots, H_{i_m})$, where $\{i_1, i_2, \ldots, i_m\} = \mathcal{I}$, and for all $j_1, j_2 \in \{1, 2, \ldots, m\}$ if $j_1 < j_2$ then $i_{j_1} < i_{j_2}$.

Definition: Let $\mathcal{S} = \{(Q_i, Act_i, \delta_i, I_i, H_i)\}_{i\in\mathcal{I}}$, where $\mathcal{I} \subseteq \mathbb{N}$ is finite, be a system of component-interaction automata such that the sets S_{H_i}, where $i \in \mathcal{I}$, are pairwise disjoint. Then $\mathcal{C} = (\Pi_{i\in\mathcal{I}} Q_i,\ \cup_{i\in\mathcal{I}} Act_i,\ \delta,\ \Pi_{i\in\mathcal{I}} I_i,\ (H_i)_{i\in\mathcal{I}})$ is a *component-interaction automaton over* $\mathcal{S}$ iff $\delta \subseteq \Delta_{\mathcal{S}}$.

Example 3.5: The CI automaton over the set $\{\mathcal{C}_i\}_{i\in\{1,2,3\}}$ can for example take form of an automaton in figure 5. In this case the selection of a transition set was motivated by the architectural assembly of a system depicted in figure 1. The architecture indicates that both *Client1* and *Client2* components may participate only in internal actions in connection with the *Database* component. Their external actions modelled by input and output transitions have to be removed from the transition space because the architecture do not enable their delegation out of the *System* component which is modelled by the composition. On contrary, the *Database* component may perform all potentially feasible actions (synchronize with *Client1*, synchronize with *Client2*, and let external actions to be delegated out of the composed system). The diagram in figure 5 depicts the resulting (reachable) transition set (solid lines) in comparison to the complete transition space (dotted lines). Every state $(q_1, q_2, q_3) \in Q_1 \times Q_2 \times Q_3$ is represented as a sequence $q_1q_2q_3$ for short.

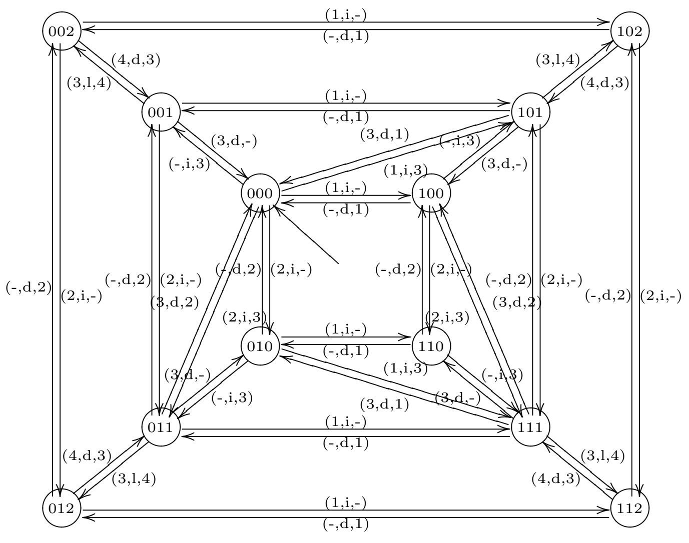
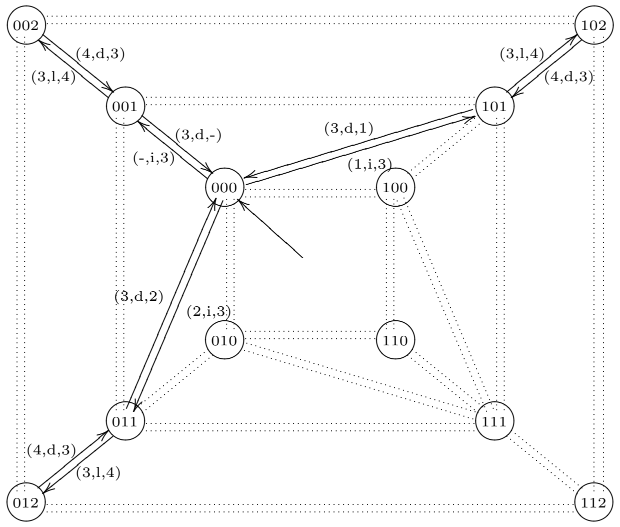


Figure 4: Complete transition space for $\{\mathcal{C}_i\}_{i\in\{1,2,3\}}$



Hierarchy of component names: $((1),(2),(3,4))$

Figure 5: CI automaton over the set $\{\mathcal{C}_i\}_{i\in\{1,2,3\}}$

4. APPLICATION AND FUTURE WORK

The Component-Interaction automata language allows for a realistic specification of component interactions in component-based systems with respect to the assembly of a system, and preserving information about the system hierarchy and primitive components communicating through the actions. That provides a rich base for application of a variety of formal methods.

One of the classical applications of formal methods is a checking of various temporal properties of system behaviour. If the system is modelled as a component-interaction automaton, the behaviour capturing the interaction among components and architectural levels is captured in (infinite) traces. Both linear and branching time temporal logics have proved to be useful for specifying properties of traces. There are several formal methods for checking that a model of the design satisfies a given specification. Among them those based on automata are especially convenient for our model of Component-Interaction automata. We have experimentally verified several specifications of component-based systems modelled as component-interaction automata with the help of a DiVinE model checking tool [1] that supports verification of LTL properties. Nowadays, we examine which (temporal) logics could be useful to specify interesting interaction properties.

One of the other issues we are currently addressing is the definition of refinement relation stating the relationship between component specification and implementation. We also study the substitutability of two components and how it can influence the properties of an overall system. Other interesting issue concerns the varieties of system assemblies and questions addressing whether there exists any assembly of the primitive components for which the composed system satisfies particular properties or which interaction properties are satisfied in every correct assembly.

5. CONCLUSION

The paper presents a formal verification-oriented automata-based specification language named Component-Interaction automata. The language aims to provide a direct and desirable way of modelling component interactions in component-based systems which is meant to be transparent and understandable thanks to the primary purpose oriented to component-based systems and their specifics. The model is inspired by some features of previously mentioned models and differs in many others. It supports freedom of choosing the transition set what allows the adjustability according to the architecture description (inspired by Team automata) and is based on synchronization of one input and one output action with the same name which becomes internal later on (inspired by Interface automata).

The model is designed to preserve all important interaction properties to provide a rich base for further verification. As a distinct from the discussed models (I/O automata, Interface automata, Team automata), it naturally preserves information about the components which participated in the synchronization and about the hierarchical structure, directly without renaming that would make the model less readable and understandable.

6. REFERENCES

[1] Divine – Distributed Verification Environment. http://anna.fi.muni.cz/divine.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, School of Computer Science, May 1997.

[3] M. Beek, C. Ellis, J. Kleijn, and G. Rozenberg. Synchronizations in Team Automata for Groupware Systems. *Computer Supported Cooperative Work—The Journal of Collaborative Computing*, 12(1):21–69, 2003.

[4] L. Brim, I. Černá, P. Vařeková, and B. Zimmerova. Component-Interaction automata as a verification-oriented component-based system specification. In *Proceedings of SAVCBS'05*, pages 31–38, Lisbon, Portugal, September 2005.

[5] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.

[6] L. de Alfaro and T. A. Henzinger. Interface-based design. In *Proceedings of the 2004 Marktoberdorf Summer School*. Kluwer, 2004.

[7] C. Ellis. Team Automata for Groupware Systems. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge (GROUP'97)*, pages 415–424. ACM Press, New York, 1997.

[8] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[9] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of PODC*, pages 137–151, April 1987.

[10] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC'95)*, September 1995.

[11] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1)*, February 1999.

[12] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.

Application of Rewriting Techniques to Verification Problems

Adam Koprowski

Technical University of Eindhoven
Department of Computer Science
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
A.Koprowski@tue.nl

Abstract. The goal of the project is to employ techniques from term rewriting to verification problems. The relationship between liveness properties and termination of term rewrite systems (TRSs) is of particular interest. The emphasis is on the investigation of such properties for infinite state space systems where standard model checking techniques fail. Next to developing the necessary underlying theory and performing a case study analysis, the possibility to automate this approach is of great importance. In this paper we discuss the motivation of such work, present the results obtained so far, discuss related work and present plans for the further research.

1 Motivation

The problem of proving termination of term rewrite systems has been studied extensively and still attracts a lot of attention of term rewriting community. Although in general undecidable, for a broad spectrum of term rewrite systems encountered in practice termination can be proven and a number of techniques has been developed to serve this goal.[1]

Apart from the theoretical results, possibility to automate the process of proving termination of TRSs was always an important issue. A number of tools for proving termination of TRSs in a fully automated manner has been developed by different authors and an annual termination competition is being organized to stimulate further work and improvement in that area.[2]

Now our main motivation is to make use of the aforementioned results and develop a method to prove liveness properties by means of TRS termination. The important point is that particular infinite state space models can be encoded by means of finite TRSs. Now, since termination proofs do not depend in any way on exploration of a state space, such problems can be tackled by our approach, whereas standard model checking approach fails for them. We will present the motivating example to illustrate our approach in Section 3.4.

[1] For an overview of term rewriting in general, and termination of term rewriting in particular, reader is refereed to, for instance, [5].

[2] See http://www.lri.fr/~marche/termination-competition for more details.

2 Related work

The idea of transforming verification problems to problems of termination of TRSs goes back to Giesl and Zantema. In [1] they presented two such transformations. The first one is sound and complete, that is a termination of the transformed TRS is equivalent to a liveness question at hand. The second one is only sound but significantly simpler. In [2] a slightly different setting has been discussed.

As already remarked in [1] the sound and complete transformation presented there is by far too complicated to be of practical use – even for simple input systems termination of transformed TRSs is difficult to show. On the other hand the preliminary case study conducted by authors revealed that the sound transformation from [1] is often not strong enough and results in non-terminating TRSs for problems where liveness do hold. That was the motivation for seeking an alternative transformation; more powerful, but still suitable for automatic termination provers. We will present such a transformation in Section 3.2.

3 Preliminary results

After giving some preliminaries in Section 3.1 we present the transformation from liveness problems to (relative) termination problems of TRSs (3.2). Then in 3.3 we describe TPA – a tool developed by authors for solving such (relative) termination problems. In Section 3.4 we illustrate our approach with an example.

3.1 Preliminaries

For a signature Σ and a set of variables $\mathcal{V}$, we denote the set of terms over Σ and $\mathcal{V}$ by $\mathcal{T}(\Sigma, \mathcal{V})$. We denote the set of variables occurring in a term t by $\mathsf{Var}(t)$. A *rewrite rule* is a pair (ℓ, r), written $\ell \to r$, with $\ell, r \in \mathcal{T}(\Sigma, \mathcal{V})$, $\ell \notin \mathcal{V}$, $\mathsf{Var}(r) \subseteq \mathsf{Var}(\ell)$. A *term rewriting system* (TRS) is a set of rewrite rules. The *rewrite relation* $\to_R$ for a TRS R is defined by $s \to_R t$ if there exists a rewrite rule $\ell \to r \in R$, a substitution δ and a context C such that $s = C[\ell\delta]$ and $t = C[r\delta]$. A TRS R is called terminating ($\mathrm{SN}(R)$[3]) if there is no infinite reduction $t_1 \to_R t_2 \to_R \ldots$.

For two relations R, S we define $R/S \equiv S^* \cdot R \cdot S^*$. We call an infinite $R \cup S$ reduction *fair* with respect to R if it contains infinitely many R-steps. The *relative termination* problem is to decide given two TRSs R, S whether $\mathrm{SN}(R/S)$. Note that $\mathrm{SN}(R/S)$ is equivalent to lack of infinite $\to_R \cup \to_S$ reductions fair with respect to $\to_R$.

Let top be a fresh unary symbol in Σ ($\mathsf{top} \notin \Sigma$). A term $t \in \mathcal{T}(\Sigma \cup \{\mathsf{top}\}, \mathcal{V})$ is called a *top term* if it contains exactly one instance of the top symbol, at the root of the term. We denote the set of top terms by $\mathcal{T}_{top}(\Sigma, \mathcal{V})$. A TRS over $\Sigma \cup \{\mathsf{top}\}$ is called a *top term rewrite system* (top TRS) if for all its rules $\ell \to r$ either both ℓ and r are top terms (*top rule*) or both ℓ and r do not contain an instance of the top symbol (*non-top rule*).

[3] It is usual to write $\mathrm{SN}(R)$ instead of $\mathrm{SN}(\to_R)$.

3.2 Transformation from liveness problems to termination problems

We make a concise presentation of the underlying theory and then present the transformation. For more elaborate description we refer the reader to [3] and [4].

We extend the notion of liveness as considered in [1] by introducing *fairness*. We define liveness with respect to a set of states S, two relations modelling computations $\rightarrow, \overset{=}{\rightarrow} \subseteq S \times S$, a set of initial states $I \subseteq S$ and a set of good states $G \subseteq S$ denoted as $\text{Live}(I, \rightarrow, \overset{=}{\rightarrow}, G)$ to hold iff:

$$\forall t_1, t_2, \ldots : \left\{ \begin{array}{r} t_1 \in I \\ \forall i : t_i \rightarrow t_{i+1} \vee t_i \overset{=}{\rightarrow} t_{i+1} \\ \forall i \exists j > i : t_j \rightarrow t_{j+1} \end{array} \right\} \implies \exists i : t_i \in G$$

Now we represent the computation states by terms, so S becomes $\mathcal{T}(\Sigma, \mathcal{V})$ and $I, G \subseteq \mathcal{T}(\Sigma, \mathcal{V})$. Abstract reduction relations $\rightarrow$ and $\overset{=}{\rightarrow}$ now correspond to rewrite relations of two TRSs over the same signature Σ: R and $R^=$, respectively. As a shorthand for $\rightarrow_R$ we write $\rightarrow$ and for $\rightarrow_{R^=}$ we simply write $\overset{=}{\rightarrow}$. Just like it is usual to write $\text{SN}(R)$ rather than $\text{SN}(\rightarrow_R)$, we will write $\text{Live}(I, R, R^=, G)$ rather than $\text{Live}(I, \rightarrow_R, \rightarrow_{R^=}, G)$.

Given some set of terms P we are going to restrict to the set of good states being terms not containing an instance of some term from P (we will denote this set by $G(P)$). Now we are going to investigate liveness properties of the form: $\text{Live}(\mathcal{T}_{top}(\Sigma, \mathcal{V}), R, R^=, G(P))$ for some top TRSs R and $R^=$. This is equivalent to proving that every infinite fair reduction of top terms contains a term which does not contain an instance of any of the terms from P. Now we will present a transformation that relates this problem with the (relative) termination of transformed systems.

Definition 1 (LT) *Let R and $R^=$ be top TRSs over $\Sigma \cup \{\mathsf{top}\}$ and $P \subseteq \mathcal{T}(\Sigma, \mathcal{V})$. The transformed systems $\text{LT}(R)$ and $\text{LT}^=(R^=, P)$ over $\Sigma \cup \{\mathsf{top}, \mathsf{ok}, \mathsf{check}\}$ are defined as follows:*

$$\boxed{\text{LT}(R)}$$

$\ell \rightarrow r$		*for all non-top rules $\ell \rightarrow r$ in R*
$\mathsf{top}(\mathsf{ok}(\ell)) \rightarrow \mathsf{top}(\mathsf{check}(r))$		*for all top rules $\mathsf{top}(\ell) \rightarrow \mathsf{top}(r)$ in R*

$$\boxed{\text{LT}^=(R^=, P)}$$

$\ell \rightarrow r$	*for all non-top rules $\ell \rightarrow r$ in $R^=$*
$\mathsf{top}(\mathsf{ok}(\ell)) \rightarrow \mathsf{top}(\mathsf{check}(r))$	*for all top rules $\mathsf{top}(\ell) \rightarrow \mathsf{top}(r)$ in $R^=$*
$\mathsf{check}(p) \rightarrow \mathsf{ok}(p)$	*for all $p \in P$*
$\mathsf{check}(f(x_1, \ldots, x_n)) \rightarrow f(x_1, \ldots, \mathsf{check}(x_i), \ldots, x_n)$	*for all $f \in \Sigma$ of arity $n \geq 1, 1 \leq i \leq n$*
$f(x_1, \ldots, \mathsf{ok}(x_i), \ldots, x_n) \rightarrow \mathsf{ok}(f(x_1, \ldots, x_n))$	*for all $f \in \Sigma$ of arity $n \geq 1, 1 \leq i \leq n$*

The following theorem from [3] relates relative termination of transformed systems with the liveness problem they originated from.

Theorem 2 (Soundness) *Let $R, R^{=}$ be top TRSs over $\Sigma \cup \{\mathsf{top}\}$, let $P \subseteq \mathcal{T}(\Sigma, \mathcal{V})$. Then:*

$$\mathrm{SN}(\mathrm{LT}(R)/\,\mathrm{LT}^{=}(R^{=}, P)) \implies \mathrm{Live}(\mathcal{T}_{top}(\Sigma, \mathcal{V}), R, R^{=}, G(P))$$

It is worth noting that under some mild additional restrictions our transformation is also complete. For details we again refer to [3].

3.3 Proving (relative) termination automatically

In the preceding section we saw how to transform liveness problems to (relative) termination problems. To deal with such problems the first author developed a tool, TPA (Termination Proved Automatically), that aims at solving such problems in an automated way. It is the first tool that also supports relative termination of TRSs, which was one of the main motivations to develop it. It uses a number of termination proving techniques, most notably semantic labelling with natural numbers, which, for the time being, is used by no other tool. It got 3rd place in the aforementioned termination competition in 2005. More information about TPA can be found on its web-page, `http://www.win.tue.nl/tpa`.

3.4 Example

Example 1 (Cars over a bridge). There is a road with cars going in two directions. But on their way there is a bridge which is only wide enough to permit a single lane of traffic. So there are lights indicating which side of the bridge is allowed to cross it. We want to verify the following liveness property: every car will eventually cross the bridge. For that clearly we need some assumptions about the lighting system. We want to be as general as possible so instead of assuming some particular algorithm of switching lights we just require them to change in a fair way, that is in the infinite observation of the system there must be infinitely many light switches. Also we assume that before a light switches at least one car will pass (otherwise liveness is lost as lights can change all the time without any cars passing).

This system can be modelled with a unary *top* symbol whose arguments start with a binary symbol *left* or *right* indicating which side has a green light. The arguments of *left* and *right* start with unary symbols *new* and *old* representing cars waiting to cross the bridge. The constant *bot* stands for the end of the queue. New cars are allowed to arrive at the end of the queue at any time. What we want to prove is that finally no old car remains. The top TRS modelling this system follows:

$$\begin{array}{rrcl rrcl}
(1) & \mathsf{top}(\mathsf{left}(\mathsf{old}(x), y)) & \to & \mathsf{top}(\mathsf{right}(x, y)) & (6) & \mathsf{top}(\mathsf{right}(x, \mathsf{bot})) & \to & \mathsf{top}(\mathsf{left}(x, \mathsf{bot})) \\
(2) & \mathsf{top}(\mathsf{left}(\mathsf{new}(x), y)) & \to & \mathsf{top}(\mathsf{right}(x, y)) & (7) & \mathsf{top}(\mathsf{left}(\mathsf{old}(x), y)) & \overset{=}{\to} & \mathsf{top}(\mathsf{left}(x, y)) \\
(3) & \mathsf{top}(\mathsf{right}(x, \mathsf{old}(y))) & \to & \mathsf{top}(\mathsf{left}(x, y)) & (8) & \mathsf{top}(\mathsf{left}(\mathsf{new}(x), y)) & \overset{=}{\to} & \mathsf{top}(\mathsf{left}(x, y)) \\
(4) & \mathsf{top}(\mathsf{right}(x, \mathsf{new}(y))) & \to & \mathsf{top}(\mathsf{left}(x, y)) & (9) & \mathsf{top}(\mathsf{right}(x, \mathsf{old}(y))) & \overset{=}{\to} & \mathsf{top}(\mathsf{right}(x, y)) \\
(5) & \mathsf{top}(\mathsf{left}(\mathsf{bot}, y)) & \to & \mathsf{top}(\mathsf{right}(\mathsf{bot}, y)) & (10) & \mathsf{top}(\mathsf{right}(x, \mathsf{new}(y))) & \overset{=}{\to} & \mathsf{top}(\mathsf{right}(x, y)) \\
 & & & & (11) & \mathsf{bot} & \overset{=}{\to} & \mathsf{new}(\mathsf{bot})
\end{array}$$

We have the following semantics of the rules: (1) – (4) car passes and the light changes; (5) – (6) : no car waiting, light can change; (7) – (10) car passes, light remains the same; (11) New car arriving.

By using our approach, in a way described in the preceding sections, the liveness property stating that every old car can eventually cross the bridge, can be transformed to a question of relative termination of TRS. This question, in turn, can be positive answered in a fully automated way by the use of TPA .

4 Conclusions and further research

We presented a framework for verification of liveness properties, that can work also for infinite state space systems. This method requires the model of the system to be given as TRS but then the proof that liveness property holds is delivered automatically by TPA by first transforming the TRS and then proving termination of the transformed TRS.

Clearly this is just the beginning of the journey and a lot of extensions is possible, among which the following ones we find particularly interesting and worth further investigation:

- Clearly our definition of fairness and of liveness problems we are aiming at could enjoy some generalization. Of particular interest would be the direction allowing us to deal not only with liveness but also with safety properties.
- Once the framework is mature enough it would be interesting to perform a case study analysis, on real-life examples.
- Some techniques for proving termination generalize to relative termination and as such they are used in TPA. But, since relative termination plays an important role in dealing with liveness with fairness, we believe that further development of relative termination techniques would be of great interest.
- As soon as such techniques are available implementing them in TPA would be a natural next step, increasing the applicability of the tool.

References

1. Jürgen Giesl and Hans Zantema. Liveness in rewriting. In *Proc. 14th RTA, LNCS 2706*, pages 321–336, 2003.
2. Jürgen Giesl and Hans Zantema. Simulating liveness by reduction strategies. *Electr. Notes Theor. Comput. Sci.*, 86(4), 2003.
3. Adam Koprowski and Hans Zantema. Proving liveness with fairness using rewriting. In *Frontiers of Combining Systems, 5th International Workshop, FroCoS 2005, Vienna, Austria, September 19-21, 2005, Proceedings*, volume 3717 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2005.
4. Adam Koprowski and Hans Zantema. Proving liveness with fairness using rewriting. Technical Report CSR 05-06, Eindhoven University of Technology, Eindhoven, The Netherlands, March 2005.
5. TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.