

Proceedings of Formal Methods 2009 Doctoral Symposium

November 6, 2009

Eindhoven, The Netherlands

MohammadReza Mousavi and Emil Sekerinski, Editors

Preface

Welcome to the third Doctoral Symposium of the International Symposium on Formal Methods. This year, the Formal Methods Symposium and its Doctoral Symposium are organized in Eindhoven, the Netherlands. It is also part of the Formal Methods Week featuring a number of scientific events dedicated to Formal Methods and their application.

The call for papers for the Doctoral Symposium was sent out in July 2009 and has attracted 20 papers from 13 different countries. The review committee then spent about one month reviewing the submitted papers and discussing them. The final decision was a particularly difficult one since 14 out of 20 papers received an average positive score from the reviewers; hence, many good submissions had to be rejected due to the limited time dedicated to the symposium and to guarantee sufficient room for discussion for the accepted papers. Finally, the review committee accepted 10 papers from 6 different countries, which are presented in this proceedings.

We would like to thank several people and organizations which helped us in organizing this symposium. First and foremost, we would like to acknowledge the help and support provided by the FM 2009 organization committee and program co-chairs: Tijn Borghuis, Erik de Vink, Jos Baeten, Ana Cavalcanti and Dennis Dams. We are grateful to Formal Methods Europe association for providing generous travel grants and free tickets to the conference dinner for the participating students. Also, we would like to thank our Review and Examination Committees, as well as the additional sub-referees for their time and effort in reviewing and selecting among the submitted papers. Our best thanks go to Professor Cliff B. Jones for accepting our invitation to give an invited talk in this symposium. Finally we would like to thank the students who have submitted to and participated in the Doctoral Symposium, without whom this event would not even materialize.

September 2009

MohammadReza Mousavi and Emil Sekerinski
Co-Chairs of FM 2009 Doctoral Symposium

Organization

Review Committee

S. Arun-Kumar (IIT, India)
Paulo Borba (UFPE, Brazil)
Michael Butler (Southampton, UK)
Jin Song Dong (NUS, Singapore)
Wan Fokkink (VU, The Netherlands)
Ichiro Hasuo (Kyoto, Japan)
Anna Ingólfssdóttir (Reykjavik, Iceland)
Joost-Pieter Katoen (RWTH, Germany)
Ian Mackie (Sussex, UK)
MohammadReza Mousavi (Eindhoven, The Netherlands)
Mila Dalla Preda (Verona, Italy)
Emil Sekerinski (McMaster, Canada)
Sandeep Shukla (VT, USA)
Bernd-Holger Schlingloff (Humboldt U. Berlin, Germany)
Elena Troubitsyna (Abo Akademi, Finland)
Tarmo Uustalu (Inst. of Cybernetics, Estonia)
Frits Vaandrager (Nijmegen, The Netherlands)
Husnu Yenigun (Sabanci, Turkey)

Examination Committee

Wan Fokkink (VU, The Netherlands)
Cliff B. Jones (Newcastle, UK)
Joost-Pieter Katoen (RWTH, Germany)
MohammadReza Mousavi (Eindhoven, The Netherlands)
Emil Sekerinski (McMaster, Canada)
Frits Vaandrager (Nijmegen, The Netherlands)

Additional Referees

Jan Calta	Robert Hilbrich	Matthias Raffelsieper
Pieter Cuijpers	Hartmut Lackner	Michel Reniers

Program

Process Algebra

09:00 - 09:35 Exploiting Architectural Constraints and Branching Bisimulation
Equivalences in Component-Based Systems. Christian Lambertz (U. Mannheim)

09:35 - 10:10 Using CSP for Software Verification. Moritz Kleine (TU Berlin)

10:10 - 10:30 Tea/Coffee Break

Formal Verification

10:30 - 11:05 Verification Architectures for Real-time Systems.
Johannes Faber (U. Oldenburg)

11:05 - 11:40 A Formal Model to Develop and Verify Self-Adaptive Systems.
Narges Khakpour (Tarbiat Modares U.)

Security

11:40 - 12:15 Rigorous Development of Java Card Applications With the B Method.
Bruno Gomes (U. Federal do Rio Grande do Norte)

12:15 - 12:50 Exploiting Loop Transformations for the Protection of Software.
Enrico Visentini (U. Verona)

12:50 - 14:00 Lunch Break

Invited Talk

14:00 - 14:30 How to do research in Formal Methods. Cliff B. Jones (Newcastle U.)

Testing

14:30 - 15:05 Towards a Theory for Timed Symbolic Testing.
Sabrina von Styp (RWTH Aachen)

15:05 - 15:40 Testing of Hybrid Systems using Qualitative Models.
Harald Brandl (Graz U. of Tech)

15:40- 16:00 Tea/Coffee Break

Specification and Refinement

16:00 - 16:35 Formal Domain Modeling: From Specification to Validation. Atif Mashkoor
(LORIA)

16:35 - 17:10 Expressing KAOS Goal Models with Event-B. Abderrahman Matoussi (U.
Paris Est)

Table of Contents

Exploiting Architectural Constraints and Branching Bisimulation Equivalences in Component-Based Systems	1
<i>Christian Lambertz</i>	
Using CSP for Software Verification	8
<i>Moritz Kleine</i>	
Verification Architectures for Real-time Systems	14
<i>Johannes Faber</i>	
A Formal Model to Develop and Verify Self-Adaptive Systems	20
<i>Narges Khakpour</i>	
Exploiting Loop Transformations for the Protection of Software	26
<i>Enrico Visentini</i>	
Rigorous Development of Java Card Applications With the B Method . . .	32
<i>Bruno Gomes</i>	
Towards a Theory for Timed Symbolic Testing	39
<i>Sabrina von Styp-Rekowski</i>	
Testing of Hybrid Systems using Qualitative Models	46
<i>Harald Brandl</i>	
Formal Domain Modeling: From Specification to Validation	53
<i>Atif Mashkoor</i>	
Expressing KAOS Goal Models with Event-B	60
<i>Abderrahman Matoussi</i>	

Exploiting Architectural Constraints and Branching Bisimulation Equivalences in Component-Based Systems

Christian Lambertz*

Department of Computer Science, University of Mannheim, Germany
lambertz@informatik.uni-mannheim.de

Abstract. We introduce a condition for the verification of properties in component-based systems that allows for components with variants and hence supports reusability. Thereby, the architecture of the system and behavior equivalences between the components are exploited in order to cope with the state space explosion problem. We use the model of interaction systems as the component model and extend it with unobservable behavior and a new composition operator.

1 Introduction

In component-based systems the fault-free composition of the components plays an important role in order to construct fault-free systems. But, the well-known state space explosion problem, that arises in the composition step, makes a direct state space analysis unfeasible. Thus, many techniques were developed that reduce the state space, e.g., compositional reasoning, partial order reduction, abstract interpretation, and symmetry reduction. Another approach is to exploit the architecture of the composed system.

The exploitation of the architecture and of behavioral equivalences is used as such a technique by Bernardo et al. [1]. They introduce the notion of component compatibility which means that the composite behavior of two components, where any joint action is concealed, is weak bisimilar to the behavior of one of the components with corresponding concealed actions. For instance, if a component b which only interacts with one component m is compatible to m , then b is not important for the global behavior of the system. Instead of considering both components one can safely consider only m .

This approach is extensible to the whole system. It is particularly suited for acyclic architectures, such as star-like or tree-like ones. For instance in star-like architectures, the compatibility check is performed for every border component together with the central component. If it succeeds, the behavior of the whole system can be reduced by only considering the behavior of the central component.

* I thank my advisor Mila Majster-Cederbaum for proposing this research direction.

A similar idea is used by Hennicker et al. [5]. They call the compatibility behavior neutrality and provide a reduction strategy for such neutral components.

However, as shown by our example in Sec. 3, the mentioned strategies are not suited for versatile components that offer many variants for different contexts. This versatility supports the reusability property that is typically desired in component-based system design. Similarly arguing, the compatibility assumption of [1] is rather restrictive. If a border component alters the behavior of the central component but this alteration is not important for the interaction with the other components, the approach already fails.

Our contribution to overcome the versatility problems of the existing approaches is that we propose a condition that allows for more liberal alterations. We use interaction systems by Gössler and Sifakis [4] as the component model. Note that our technique does not rely on this model; our ideas carry over to other models of concurrency as well. The key idea of interaction systems is the separation of the components' description and their glue code. Each component's description consists of a static and a dynamic part. The static part describes the available actions for cooperation, and the dynamic part models the local behavior as a labeled transition system. The glue code is described as a set of so-called interactions and models the cooperation. The global behavior can be computed by combining the local behaviors according to the glue code. We extend the original definition of [4] by the concept of closed interactions that cannot be used for further compositions and that become unobservable in the global behavior. Additionally, we provide a new composition operator.

In the following, we focus on component systems with star-like and tree-like architectures. This is reasonable since many interesting cases, e.g., master/client architectures, follow such a pattern. In CSP, for instance, the connection diagram of processes constructed with the subordination operator forms a tree [6].

Furthermore, we use branching bisimilarity (denoted by \approx_b in the following) instead of weak bisimilarity (denoted by \approx), which is used by Bernardo et al. [1] and Hennicker et al. [5], because branching bisimilarity preserves more properties of systems (a logical characterization of \approx_b in CTL*-X exists [2]), it is more efficient to calculate, and, as remarked by van Glabbeek and Weijland [3], many system that are weak bisimilar are also branching bisimilar.

2 Definitions

Before we give a precise definition of interaction systems, we define several operators for the manipulation of sets that are needed in the following.

Definition 1 (Set Manipulation). *Let X and Y be two sets of sets with $X \cap Y = \emptyset \vee X \cap Y = \{\emptyset\}$. The (nonempty) interjoin of these sets is denoted by $X \bowtie Y := \{x \cup y \mid x \in X \wedge y \in Y\} \setminus (\{\emptyset\} \cup X \cup Y)$, i.e., the interjoin contains only new sets that were not contained in X or Y before. The power set of a set x is denoted by $\wp(x) := \{x' \mid x' \subseteq x\}$. We overload this operator for sets of sets, i.e., the union of all power sets of sets x contained in X is denoted by*

$\wp(X) := \bigcup_{x \in X} \wp(x)$. The set of sets X is said to respect the sets in Y , denoted by $X \sqsubseteq Y$, if $\forall x \in X \exists y \in Y: x \subseteq y$.

Definition 2 (Interaction System). An interaction system is defined by means of a tuple $Sys := (K, \mathcal{A}, Int, Int_{closed}, \{T_i\}_{i \in K})$. Here K is a finite set of components, which are referred to as $i \in K$. The actions of component i are given by the action set A_i with the property $\forall i, j \in K: i \neq j \Rightarrow A_i \cap A_j = \emptyset$. All available actions are contained in the global action set $\mathcal{A} := \bigcup_{i \in K} A_i$.

A nonempty finite set $\alpha \subseteq \mathcal{A}$ of actions is called an interaction if it contains at most one action of every component, i.e., $|\alpha \cap A_i| \leq 1$ for all $i \in K$. For any interaction α and component i we put $i(\alpha) := A_i \cap \alpha$. We say that i participates in α if $i(\alpha) \neq \emptyset$.

The interaction set Int of Sys is a set of interactions that covers all actions, i.e., $\bigcup_{\alpha \in Int} \alpha = \mathcal{A}$. The set of closed interactions $Int_{closed} \subseteq Int$ contains interactions that cannot be used to create new interactions among the components (we define later how this creation works). An additional special interaction, the unobservable interaction τ , is available but not contained in the global interaction set, i.e., $\tau \notin Int$.

Finally, for each component i a labeled transition system T_i describes the local behavior of i , i.e., $T_i := (Q_i, A_i, \Delta_i, q_i^0)$ where Q_i is the local state space, each local alphabet A_i contains all actions of component i , the local transition relation is $\Delta_i \subseteq Q_i \times A_i \times Q_i$, and $q_i^0 \in Q_i$ is the local initial state.

The global behavior of Sys is a labeled transition system $\llbracket Sys \rrbracket := (Q, A, \Delta, q^0)$ which is obtained in a straightforward manner. The global state space is given by $Q := \times_{i \in K} Q_i$. States are denoted by tuples $q := (q_1, \dots, q_n)$, and the global initial state is $q^0 := (q_1^0, \dots, q_n^0)$. The global alphabet $A := Int \setminus Int_{closed} \cup \{\tau\}$ contains all non-closed interactions and the special interaction τ . The global transition relation $\Delta \subseteq Q \times A \times Q$ is defined canonically: For any $\alpha \in Int$ and any $q, q' \in Q$ we have

- $(q, \alpha, q') \in \Delta$ if $\alpha \notin Int_{closed}$ and $\forall i \in K: \text{if } i(\alpha) = \{a_i\} \text{ then } (q_i, a_i, q'_i) \in \Delta_i \text{ and if } i(\alpha) = \emptyset \text{ then } q_i = q'_i \text{ and}$
- $(q, \tau, q') \in \Delta$ if $\alpha \in Int_{closed}$ and $\forall i \in K: \text{if } i(\alpha) = \{a_i\} \text{ then } (q_i, a_i, q'_i) \in \Delta_i \text{ and if } i(\alpha) = \emptyset \text{ then } q_i = q'_i$.

Next, we define a composition operator for interaction systems. This composition should only be possible for disjoint interaction systems: Two interaction systems Sys and Sys' are *disjoint*, if their set of components and global action sets are disjoint.

Definition 3 (Composition: Interconnecting Interaction Systems). Let Sys and Sys' be two disjoint interaction systems, and let $I^+ \subseteq \wp(Int \setminus Int_{closed}) \boxtimes \wp(Int' \setminus Int'_{closed})$ be a set of new interactions. Let $I^- \subseteq (Int \setminus Int_{closed}) \cup (Int' \setminus Int'_{closed})$ with $I^- \sqsubseteq I^+$ be a set of old interactions that should not be included in the composite interaction system because any of these old interactions is part of a new interaction. Let (I^+, I^-) denote this composition information. The composition of Sys and Sys' with respect to the composition information

is the interaction system $Sys \underset{(I^+, I^-)}{\otimes} Sys' := (K \cup K', \mathcal{A} \cup \mathcal{A}', (I^+ \cup Int \cup Int') \setminus I^-, Int_{closed} \cup Int'_{closed}, \{T_i\}_{i \in K \cup K'})$.

We now define subsystems of interaction systems by considering subsets of the components. Afterwards, we define an operator for declaring interactions as closed.

Definition 4 (Subsystem Construction). *Let Sys be an interaction system and $K_1 \subseteq K$ a set of components. The subsystem of Sys obtained by only considering the components in K_1 , denoted by $Sys[K_1]$, is the interaction system $Sys[K_1] := (K_1, \mathcal{A}[K_1], Int[K_1], Int_{closed}[K_1], \{T_i\}_{i \in K_1})$ with $\mathcal{A}[K_1] := \bigcup_{i \in K_1} A_i$, $Int[K_1] := \{\alpha \cap \mathcal{A}[K_1] \mid \alpha \in Int \wedge \alpha \cap \mathcal{A}[K_1] \neq \emptyset\}$, and $Int_{closed}[K_1] := \{\alpha \in Int_{closed} \mid \alpha \subseteq \mathcal{A}[K_1]\}$.*

Definition 5 (Closing of Interactions). *Let Sys be an interaction system and \hat{I} be a set of interactions. The closing of the interactions contained in \hat{I} in Sys , denoted by $Sys \parallel \hat{I}$, is the interaction system $Sys \parallel \hat{I} := (K, \mathcal{A}, Int, Int_{closed} \cup (\hat{I} \cap Int), \{T_i\}_{i \in K})$.*

In order to avoid confusing parentheses when the three operators are used together, we define the following order of operators: In the absence of parentheses, subsystem construction takes precedence over closing, which takes precedence over composition.

In the following, we focus our analyses on systems with a particular architecture: The *interaction graph* of an interaction system Sys contains a node for every component and the set of edges $\{\{i, j\} \mid \exists \alpha \in Int \text{ such that components } i \text{ and } j \text{ participate in } \alpha\}$. If the interaction graph forms a tree in the graph-theoretical sense we say that Sys is *tree-like*. If it contains exactly one inner node we say that Sys is *star-like*.

Note that the above definition of tree/star-like interaction systems implies that all interactions are binary. Furthermore, we assume for simplicity that any tree-like interaction system that is considered in the following satisfies the *exclusive communication* property, i.e., any action of any component is only contained in interactions with exactly one other component. This requirement does not restrict our approach, because an arbitrary tree-like interaction systems Sys can be transformed into an equivalent tree-like interaction system Sys' with exclusive communication in polynomial time [7].

3 Motivation: Versatile Merchandise Management System

Consider a merchandise management system (MMS) for wholesalers which manages orders of customers and supplies in the wholesaler's storage. Therefore, the MMS offers several modes of operation: A wholesaler may deliver after receiving an order, may demand from the customer to ask for a reservation before ordering, and may accept direct orders but request nevertheless a reservation for

internal purposes. Additionally, any reservation is printed out for internal use. This versatile behavior is reasonable if we assume that the MMS was developed by a software company that wants to sell the MMS as a software component to a variety of wholesalers. We consider now a particular wholesaler that bought and uses this MMS who has a storage system in which every product needs to get reserved before it will be delivered. Additionally, this wholesaler requests from its customers to ask for a reservation before placing an order. This setting is modeled as an interaction system with three components representing the MMS m , the storage system s , and a customer c . The behavior of the components is depicted in Fig. 1. Obviously, this interaction system is star-like.

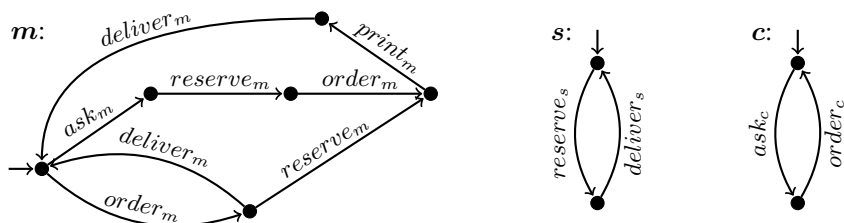


Fig. 1. Behavior of the components: the MMS m , the storage system s , and the customer c .

The interactions are given by $Int := \{\{reserve_m, reserve_s\}, \{deliver_m, deliver_s\}, \{ask_m, ask_c\}, \{order_m, order_c\}, \{print_m\}\}$. Note that many more customers could be added to the system, e.g., the customers are classified into several groups. Furthermore, many more storage systems could be added, e.g., distributed storage places exist each with an own system. Of course, the star-like architecture is preserved this way.

We want to verify the deadlock-freedom of the system. If we want to use the approach of Bernardo et al. [1] or Hennicker et al. [5], we have to check whether the behavior of the subsystem consisting of either a storage system or a customer composed with the MMS is weak bisimilar to the behavior of the MMS where any action used in an interaction in the former system is closed. We need to check all such pairs.

Stated in our notation, we need to check for $i := s, c$ whether $\llbracket Sys[\{m, i\}] \parallel \hat{I}_{m,i} \rrbracket \stackrel{?}{\approx} \llbracket Sys[\{m\}] \parallel \hat{I}_{m,i} \rrbracket$ holds where $\hat{I}_{m,i}$ denotes the interactions in which both m and i participate. This is not the case for $i = c$ as illustrated in Fig. 2, because the path “ $\tau \{deliver_m\}$ ” is only possible in the latter system.

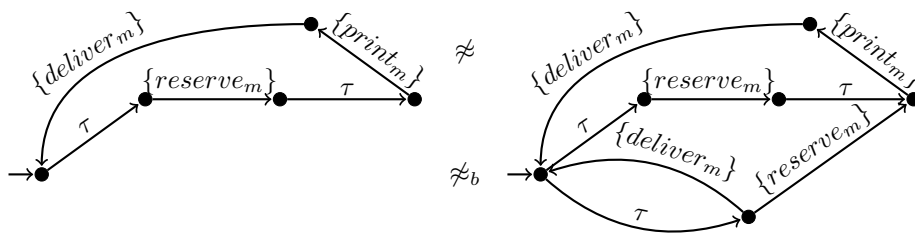


Fig. 2. Global behavior of $Sys[\{m, c\}] \parallel \hat{I}_{m,c}$ (on the left) and $Sys[\{m\}] \parallel \hat{I}_{m,c}$ (on the right).

But, this path is not possible in the global system, i.e., the restrictions that component c puts on m play no role for the interaction with component s . We can express this situation with the following equivalence:

$$\llbracket Sys[\{m, c\}] \parallel \hat{I}_{m,c} \otimes_{\langle m,s \rangle} Sys[\{s\}] \rrbracket \approx_b \llbracket Sys[\{m\}] \parallel \hat{I}_{m,c} \otimes_{\langle m,s \rangle} Sys[\{s\}] \rrbracket$$

where $\langle m, s \rangle$ denotes the composition information of m and s . Since this equivalence holds and also

$$\llbracket Sys[\{m, s\}] \parallel \hat{I}_{m,s} \otimes_{\langle m,c \rangle} Sys[\{c\}] \rrbracket \approx_b \llbracket Sys[\{m\}] \parallel \hat{I}_{m,s} \otimes_{\langle m,c \rangle} Sys[\{c\}] \rrbracket$$

holds, we can conclude that no border component restricts the central component m in a way, that the interactions of m with any other border component are interfered.

Next, we formalize the ideas behind the example and show how it can be applied to arbitrary interaction systems whose topology follows a particular architectural constraint. Afterwards, we complete the example by showing its deadlock-freedom.

4 Exploiting the Architecture and Branching Bisimilarities

Consider an interaction system Sys with a star-like architecture, i.e., one central component is surrounded by border components and each border component interacts only with the central component. Note that we have already extended our approach to tree-like systems. The following theorem is also applicable in the tree-like case because such a systems consists of many star-like subsystems. Thereby, we use the graph-theoretical center of the interaction graph as the central component of the tree-like system.

An arbitrary border component i in Sys does not interfere the central component m , if any restriction put on m by i plays no role for the interaction of m with any other border component. Of course, the border component i is allowed to restrict the behavior of the central component, but only in this non-interfering way. The idea behind the non-interference is that the border component i does not heavily influence the global behavior of the system and its behavior can be neglected in the analysis. We formalize this idea for all border components in Theorem 1. Thereby, the non-interference is modeled by branching bisimilar behavior.

Theorem 1. *Given a star-like interaction system Sys with exclusive communication. Let m denote the central component. If for all distinct pairs $i, j \in K \setminus \{m\}$ holds*

$$\llbracket Sys[\{m, i\}] \parallel \hat{I}_{m,i} \otimes_{\langle m,j \rangle} Sys[\{j\}] \rrbracket \approx_b \llbracket Sys[\{m\}] \parallel \hat{I}_{m,i} \otimes_{\langle m,j \rangle} Sys[\{j\}] \rrbracket$$

with $\hat{I}_{m,i} := \{\alpha \in Int[\{m, i\}] \mid \alpha \notin Int[\{m\}] \vee \exists \alpha' \in Int[\{m, i\}]: \alpha \subset \alpha'\}$ and $\langle m, j \rangle := (I_{m,j}^+, I_{m,j}^-)$ in each case with $I_{m,j}^+ := Int[\{m, j\}]$ and $I_{m,j}^- := \{\alpha \in Int[\{m\}] \cup Int[\{j\}] \mid \alpha \notin Int[\{m, j\}]\}$ then it holds that

$$\llbracket Sys \parallel \hat{I} \rrbracket \approx_b \llbracket Sys[\{m, k\}] \parallel \hat{I} \rrbracket$$

with $\hat{I} := \bigcup_{i \in K \setminus \{m, k\}} \hat{I}_{m, i} = \{\alpha \in \text{Int} \mid \alpha \notin \text{Int}[\{m, k\}] \vee \exists \alpha' \in \text{Int}: \alpha \subset \alpha'\}$ and an arbitrary border component $k \in K \setminus \{m\}$.

Theorem 1 shows that the global behavior of a star-like interaction system Sys , which could be too large for a direct analysis, can be minimized if certain equivalences between subsystems hold. But, the minimization is only useful if properties of Sys are preserved. Fortunately, branching bisimilarity preserves many properties [2]. The following corollary provides this preservation for the minimization.

Corollary 1. *Given a star-like interaction system Sys with exclusive communication and a property expressed as a CTL*-X formula ϕ . Let m denote the central component of Sys and k one of the border components.*

If Theorem 1 holds for Sys , i.e., $\llbracket Sys \setminus \hat{I} \rrbracket \approx_b \llbracket Sys[\{m, k\}] \setminus \hat{I} \rrbracket$ with $\hat{I} := \{\alpha \in \text{Int} \mid \alpha \notin \text{Int}[\{m, k\}] \vee \exists \alpha' \in \text{Int}: \alpha \subset \alpha'\}$, and if no interaction in which a border component but component k participates is used as an atomic proposition in ϕ , i.e., $\text{AP}(\phi) \subseteq (\text{Int} \setminus \hat{I}) \cup \{\perp\}$, then the satisfiability of ϕ in Sys is implied by the satisfiability of ϕ in $Sys[\{m, k\}]$, i.e., $Sys[\{m, k\}] \models \phi \Rightarrow Sys \models \phi$.

Completing the Example in Section 3 We want to verify the deadlock-freedom of the MMS by verifying whether the CTL*-X formula $\phi = \text{AG EF true}$ holds. We apply Theorem 1, and since it holds and $Sys[\{m, c\}]$ satisfies ϕ , the system is deadlock free.

5 Future Work

If the premises of Theorem 1 do not hold for all border components, a smaller system could be obtained by only considering a subsystem which satisfies the theorem. Additionally, special protocols which represent small parts of a component's behavior could be used to further simplify the checks. Other architectures are also under consideration.

References

1. M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM Trans. Softw. Eng. Methodol.*, 11(4):386–426, 2002.
2. R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.
3. R. J. van Glabbeek and W. P. Weijland. Branching time and abstraction in bisimulation semantics. *J. ACM*, 43(3):555–600, 1996.
4. G. Gössler and J. Sifakis. Composition for component-based modeling. In *Proceedings of FMCO '02*, pp. 443–466, 2003.
5. R. Hennicker, S. Janisch, and A. Knapp. On the observable behaviour of composite components. In *Proceedings of FACS '08*, 2008.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
7. M. Majster-Cederbaum and M. Martens. Compositional analysis of deadlock-freedom for tree-like component architectures. In *Proceedings of EMSOFT '08*, pp. 199–206, 2008.

Using CSP for Software Verification

Moritz Kleine

Technische Universität Berlin
Institute for Software Engineering and Theoretical Computer Science
Berlin, Germany
`mkleine@cs.tu-berlin.de`

Abstract. In this paper, we present our approach to verifying software by synthesizing a CSP model from its compiler intermediate representation. This allows us to reason about the implementations of concurrent programs on the CSP level and to reuse existing CSP tools. The correspondence of an implementation to its CSP-based specification can be established by proving that the synthesized CSP model is a refinement of the specification. The main contribution of this work is a new source-language-independent semi-automated approach to verifying concurrent software.

1 Introduction

This research addresses the problem of building usable software verification systems for concurrent systems implemented in a general-purpose programming language. One problem in this field is developing efficient tools and integrated tool chains supporting software verification. Additionally, still an active field of research is the development of specification languages that are suitable for software verification, e.g. Spec# [1], KeY-C [12] and VCC [2]. To facilitate the verification of concurrent programs, we propose an approach that builds on the automated synthesis of a low-level CSP model from the compiler intermediate representation (IR) of their implementation-level description. Our work explores several ways of building the low-level model, which formalizes the IR of the program and outlines different ways of exploiting the low-level model for software verification.

1.1 Brief Introduction to CSP

Communicating Sequential Processes (CSP) is a process calculus developed in the early 1980s [4]. It is capable of specifying and verifying reactive and concurrent systems, where the modeling of communication plays a key role. CSP is equipped with a rich set of process operators for defining possibly infinite transition systems by, for example, prefixing ($a \rightarrow P$), sequential composition ($P_1; P_2$), hiding ($P \setminus A$) and parallel composition ($P_1 \parallel A \parallel P_2$). The semantics of CSP processes can be given in different ways. The most popular semantics are trace semantics, failure semantics and failure-divergence semantics [13]. All these

semantics are supported by the automatic refinement checker FDR2 [4], which is one of the tools we use for verification purposes.

CSP_M is a machine-readable version of CSP that has been developed as the input language for the FDR2 tool. CSP_M extends CSP by a small but powerful functional language, which offers constructs such as lambda and let expressions and supports advanced concepts like pattern matching and currying. The language provides a number of predefined data types, e.g. booleans, integers, sequences and sets, and also allows user-defined data types. The global event set is defined by the set of typed channel declarations of a CSP_M script.

CSP_M is now the de facto standard of machine-readable CSP. Besides FDR2, the model checker and animator ProB [10] supports CSP_M , so CSP_M models can also be explored by animation and verified by LTL model checking.

1.2 Brief Introduction to LLVM

The Low Level Virtual Machine (LLVM) compiler infrastructure provides a modular framework that can be easily extended by user-defined compilation passes. It also offers a diverse set of predefined analyses and optimizations that can be used out of the box. This makes LLVM a great platform for the development of source code transformation and analysis tools. The heart of the compiler infrastructure project is its intermediate representation (IR). It is a typed assembler-like language [9], which is used internally as the basis for compiler optimizations. The LLVM framework provides gcc-based frontends for a variety of programming languages.

2 Synthesizing a Low-Level CSP Model

In [8], we sketched how to synthesize a low-level CSP model from the LLVM IR of a concurrent program. The idea underlying this approach is depicted in Fig. 1. Instead of following the classical approach of refining specifications semi-automatically down to executable code, we propose that the software engineering process begin with the development of a high-level specification in some CSP-based formalism, on the top level and that the refinement chain be cut at a level that still abstracts from implementation details. On the top level, either a CSP_M specification or a specification in an arbitrary CSP-based formalism for which a transformation into CSP_M exists, is required. It is then the programmer's job to produce efficient and robust code¹ – symbolized by arrow (1). Unlike code obtained using automatic code generators, our approach makes it more feasible to create high-performance code that meets the application's needs in terms of memory and power consumption. This procedure induces a semantic gap between the high-level specification and the final implementation, which we bridge by generating another CSP model. This model is semi-automatically

¹ Robust code is commonly understood to be not only free of bugs but also well structured, human-readable and adhere to a given set of coding guidelines so that the code is easy to maintain and extend.

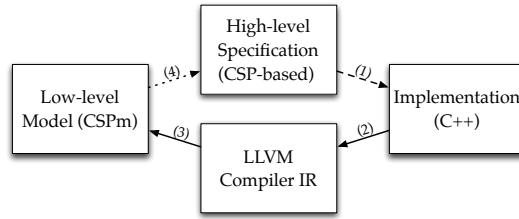


Fig. 1. Illustration of our verification methodology.

obtained from the LLVM IR, which is created during the compilation process with an LLVM-based compiler, symbolized by arrows (3) and (2), respectively. Fig. 1 also relates the generated CSP_M model to the high-level specification by arrow (4), which stands for a refinement proof. This is necessary to prove that the implementation meets its specification. Steps (3) and (4) can be used to investigate the implementation from different points of view. Especially if independent aspects have to be met by the implementation, they should be explored separately. This reduces the size of the low-level model, which is desirable for analysis by refinement and model checking.

The low-level CSP_M model contains not only processes, types and channels that are generated from the LLVM IR of a program but also two predefined parts which model platform- and domain-specific parts of the system under investigation. The platform-specific part comprises the environment model and hardware details, while the domain-specific part encompasses aspects that are common to a domain of applications, e.g. system startup and scheduling, which are provided as foundation libraries that the program builds on. These two parts are mostly manually modeled but are parameterized so that they can be reused by all applications of the domain they have been designed for. Examples of such parameters are typing information for the channels and the set of thread identifiers. The third part is the application-specific one, which describes the behavior of the threads of a multithreaded program with respect to a set of given variable names, function calls and annotations². We are currently implementing an LLVM tool that realizes the automated part of the synthesis process (arrow (3)). This tool was used to create the low-level model of the scheduler of the BOSS operating system pico-kernel, which we presented in [7].

3 Design of the Low-Level CSP Model

As discussed in the previous section, the low-level CSP model is divided into three distinct parts. The domain- and platform-specific parts are manually mod-

² Annotations can be realized using so-called ghost method and ghost variables. A ghost method is a method that modifies ghost variables only, while a ghost variable is a variable that is used for verification purposes only. Ghost code is commonly compiled into the IR for verification purposes but is not part of the final binary.

```

channel read1, write1, read2, write2, read3, read4, write3, write4 : {0,1}
V1 = let V1'(v) = read1!v -> V1'(v) [] write1?x -> V1'(x)
  within read1?x -> V1 [] write1?x -> V1'(x)
V2 = V1[[read1 <- read2, write1 <- write2]]
V3 = V1[[read1 <- read3, write1 <- write3]]
V4 = V1[[read1 <- read4, write1 <- write4]]
WithHeap(P) = (P) [|{|read1, write1, ..|}|] (V1 ||| V2 ||| V3 ||| V4)

```

Fig. 2. CSP_M model of the heap.

eled but are parameterized. The parameters and the application-specific part are synthesized from the LLVM IR of the program under consideration. Since we aim to use FDR2 for establishing the formal refinement relation between the specification and the low-level model, all models must be designed as efficiently as possible. The FDR2 manual contains a couple of rules that have to be taken into account when creating a CSP_M model to achieve the best performance with FDR2. Fig. 2 shows an efficient example of modeling a memory that stores four bit fields, constructed of parallel processes ($V1, \dots, V4$) that model a single variable each. These four processes are structurally equal, so just one of them is modeled manually ($V1$), the others being derived from it by renaming³. This model of a memory is a process, which is synchronized with the application specific part later on using the function (*WithHeap*). We use this concept to model the heap and the stacks of the threads. The process allows us to read an arbitrary value from uninitialized memory cells.

Our approach makes strong use of abstraction to reduce the size of the resulting low-level model in terms of reachable states. This includes abstracting the ranges of data types and abstracting away regions of code that do not transitively influence any of a given set of variables to be included in the low-level model. If, for example, concurrent accesses to a shared counter variable have to be proved race-condition-free, it is sufficient to build the model from the accesses to this shared counter and the locks protecting it.

The expressiveness of CSP_M imposes a limiting factor to formalizing the semantics of LLVM IR. We therefore restrict ourselves to modeling facilities that are available in CSP_M . Our approach currently supports functions, function calls, conditional and unconditional branching as well as integer arithmetic. It builds on a memory model that supports integers, arrays and uninitialized values. Depending on the properties to be proved on the models, we also use the concept of error codes to detect such sources of unwanted behavior or to signal situations that were introduced by abstractions during synthesis of the model. An error code is a fresh event $a \notin \Sigma$ and is always used in the pattern $a \rightarrow STOP$. In [8], we use this concept to detect integer overflow that was introduced by abstraction and did not indicate a real error in the low-level model. A method on the LLVM

³ One of the rules mentioned before is, for example, that renaming is to be used in preference to the parameterizing of a process definition.

IR level is translated into a CSP_M function that returns sequential processes, each modeling a single IR operation. These application-specific processes end up in a domain-specific process modeling the continuation of the application, possibly including a thread switch. Further details of the application-, domain- and platform-specific models are given in [8].

4 Analyzing the Low-Level Model

The low-level CSP model can then be either proved (or disproved) to be a refinement of its specification using the refinement checker FDR2 or animated and analyzed with the LTL model checker ProB. In [11, prop. 6], Leuschel et. al. state that the satisfaction of LTL formulas, which are limited to properties on traces, is preserved by failures refinement for finitely branching processes. Thus if one manages to generate a low-level model from the implementation level description, which is finitely-branching and a failures refinement of its specification, an LTL formula that has been proved to hold on the specification also holds on the low-level model. In this section, we clarify some issues that arise in the context of CSP refinement and LTL model checking of CSP specifications. ProB supports an LTL dialect that supports the formulation of properties on traces and refusals of processes. In addition to the temporal operators G, F, X, U and the logical connectives conjunction, disjunction and implication, ProB supports the e operator for this purpose. The expression $e(a)$ checks if the event a is enabled in the current state, for example. Prop. 6 does not extend to these kind of LTL properties, as demonstrated by the following example:

$$P = a \rightarrow P \sqcap b \rightarrow P \qquad Q = a \rightarrow Q \sqcap b \rightarrow Q$$

$$\phi = G((e(a) \Rightarrow \neg e(b)) \wedge (e(b) \Rightarrow \neg e(a)))$$

Q is a failures refinement of P , ϕ holds on P but does not hold on Q . It follows that only a subset of LTL supported by ProB can be used as long as satisfaction of LTL formulas is needed. The proof of Prop. 6 neither refers to the situation of two processes that are deadlock-free nor to specific sets of refusals except the set of all events. Thus, the only interesting situation is the one of a deadlocking state. Trace refinement with the additional requirement that the two processes are deadlock-free does preserve satisfaction of LTL formulas. Since we do not wish to limit ourselves to deadlock-free specifications, we define the notion of LTL satisfaction preserving refinement which is trace refinement and whenever a trace of the implementation cannot be extended further (it deadlocks), the same trace of the specification cannot be extended in the specification either. We plan to implement this kind of refinement as a variant of the existing trace and failure refinement checking procedures of the FDR2 tool. Another option that we are considering is switching from CSP_M to $\text{CSP}\#$ and implementing this refinement as an extension of the PAT [14] toolkit.

5 Conclusions and Future Work

In this paper, we presented a CSP-based methodology for verifying the implementations of concurrent systems. Verifying such a system requires both its abstract CSP-based specification and the LLVM IR of its implementation. Our methodology determines how significant parts of a low-level model can be synthesized from the LLVM IR of the implementation and it requires that the low-level model be a refinement of the specification. Our approach enables us to use state-of-the-art CSP tools such as FDR2 and ProB for the automated verification of concurrent programs written in a high-level programming language supported by the LLVM system. In this respect, it is source-language independent.

Instead of outputting a CSP_M script for animation, model and refinement checking, we plan to output Isabelle/HOL code, e.g. for the CSP-Prover theory [6]. Targeting an Isabelle/HOL theory would enable us to use a much more powerful type system than that of CSP_M . Additionally, it would eliminate the need to justify the abstractions introduced when reducing the ranges of the types so that model and refinement checking can be applied to the CSP_M model. To retain the automated nature of our approach, future work will also need to develop abstractions to keep the low-level models of a reasonable size as our use cases grow in code size.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*.
2. E. Cohen, M. Moskal, W. Schulte, and S. Tobies. A Practical Verification Methodology for Concurrent Programs. Microsoft Research, 2009.
3. C. Fischer. CSP-OZ: a combination of object-Z and CSP. In *FMOODS 1997*.
4. Failures-Divergence Refinement - FDR2 User Manual. <http://www.fsel.com/documentation/fdr2/fdr2manual.pdf>, 2005.
5. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
6. Y. Isobe and M. Roggenbach. A Generic Theorem Prover of CSP Refinement. In *TACAS 2005*.
7. M. Kleine, B. Bartels, T. Göthel, and S. Glesner. Verifying the Implementation of an Operating System Scheduler. In *TASE 2009*.
8. M. Kleine and S. Helke. Low Level Code Verification Based on CSP Models. In *SBMF 2009*.
9. C. Lattner and V. Adve. LLVM Language Reference Manual. <http://llvm.org/docs/LangRef.html>, 2008.
10. M. Leuschel and M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. In *ICFEM 2008*.
11. M. Leuschel, T. Massart, and A. Currie. How to make FDR Spin: LTL model checking of CSP using Refinement. In *FME 2001*.
12. O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A Tool for Verification of C Programs. In *CADE 2007*.
13. A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
14. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*.

Verification Architectures for Real-time Systems*

Johannes Faber

Department of Computing Science, University of Oldenburg, Germany
j.faber@uni-oldenburg.de

1 Introduction

In the analysis of realistic systems, one has to cope with different and heterogeneous dimensions that have to be modelled and ideally automatically verified. Real-world systems, e.g., the European Train Control System (ETCS) [1], are determined by process and communication aspects, by rich data structures, and by real-time behaviour. In [2] the ETCS system is modelled using the combined specification language CSP-OZ-DC (COD), which is designed to deal with these system dimensions; a verification approach for COD against Duration Calculus (DC) [3] formulae is provided. But unfortunately, realistic systems are most often too complex to be verifiable fully automatically. So, further decomposition methods are necessary. [2] provides an intuitive manual decomposition that splits the system and a global safety property according to an abstract behavioural protocol. It divides the system runs into several phases (e.g., braking phase, running phase, etc.) with local properties (defined as DC formulae) that hold during these phases. Once the desired property's correctness for such a protocol is established, one only has to verify that the local properties are fulfilled by the system model to guarantee correctness of the global property.

The aim of this conceptual work is to generalise this approach. We extend the specification language CSP [4] by data constraints and undefined processes and show that it is suited to specify those protocols. We introduce a sequent-style calculus over this CSP extension that allows for establishing desired properties under local real-time assumptions. All concrete specifications that are instantiations of abstract protocols and for that the local assumptions are valid automatically inherit the desired properties. With a simple proof rule (that we do not present here) it is possible to show efficiently that a concrete specification is such an instantiation. The correctness of the local assumptions can be shown using established methods for the assumptions' logic. This integration of an operational language to describe protocols and a declarative (real-time) language to describe local properties of a system to simplify verification of large systems distinguishes our approach from standard refinement/implementation approaches, e.g., CSP refinement [5] or data refinement for Z [6]. Hence, we call this combination of abstract protocol and local assumptions *Verification Architecture*.

We summarise our contributions: (1) We provide a new conceptual approach on how to use design patterns, called Verification Architectures (VA),

* This work was partly supported by the German Research Council (DFG) under grant SFB/TR 14 AVACS. See <http://www.avacs.org> for more information.

as a decomposition technique to enable verification of large systems. (2) We introduce a CSP dialect with data, undefined process parts, and local real-time assumptions for the specification of VAs. (3) A new sequent-style calculus over this CSP dialect allows for the verification of desired properties of those VAs. (4) Using a train control system motivated by the ETCS similar to the example from [2], we provide evidence that our method enables the automatic verification of a system that is too large to be verified without decomposition techniques.

The paper is structured as follows. Section 2 explains our approach formally. We introduce the CSP extension and, exemplarily, some sequent calculus proof rules in Sect. 3. Section 4 concludes with experimental results and related work.

2 General Approach

Let $va(\bar{p})$ be an abstract behavioural protocol depending on a vector of parameters \bar{p} . We will use CSP processes with data for the specification of those protocols. Additionally, we consider assumptions $asm_1(\bar{p}), \dots, asm_n(\bar{p})$ over va that also depend on the parameters – here the assumptions are dense real-time properties that are given as DC formulae. Our aim is to show that a global safety property $safe(\bar{p})$ is valid for every possible model that is an instantiation of the abstract protocol va with assumptions $asm_1(\bar{p}), \dots, asm_n(\bar{p})$.

To apply our approach, we have to show in a first step that the architecture together with the assumptions is correct for all possible parameter valuations:

$$(\forall \bar{p} \bullet va(\bar{p}) \wedge \bigwedge_{i=1, \dots, n} asm_i(\bar{p})) \models safe(\bar{p}) \quad (1)$$

This verification task to verify the correctness of the abstract parametric model va is for realistic systems not necessarily easy (in general, it cannot be done by model checking) and we will provide proof rules for the verification. But once it is verified, this result is reusable as all instantiations of this architecture inherit the correctness property automatically. We only have to show that it is an instantiation of the abstract CSP protocol and that the local assumptions asm_1, \dots, asm_n are valid, which is due to their locality easier than to verify the global property $safe$ directly. To be more concrete, we consider an instantiation $cod_C(\bar{p}_0)$ of the abstract protocol va , where \bar{p}_0 represents an instantiation of the parameters. As specification formalism, we use the parametric, combined specification language *CSP-OZ-DC* (COD) [7,8,2], since it is in line with the focused system class of complex, heterogeneous real-time systems. We now apply the result of the architecture's correctness from (1) to conclude the correctness of the concrete model cod_C . Firstly, we have to show that every trace of cod_C from the trace set $\llbracket cod_C \rrbracket$, is also a trace of va , i.e., $\llbracket cod_C \rrbracket \subseteq \llbracket va \rrbracket$. This relation can be shown syntactically for a specific class of instantiations. Thus, it is easy to verify. Secondly, we have to show that the assumptions are valid for the concrete specification:

$$cod_C(\bar{p}_0) \Rightarrow \bigwedge_{i=1, \dots, n} asm_i(\bar{p}_0) \quad (2)$$

This can be done by applying existing model checking techniques [2] for COD and DC. With this, our approach yields that the desired safety property is valid for the concrete model. We argue that this proposition is correct. From (1) we can conclude (3), due to $\llbracket cod_C \rrbracket \subseteq \llbracket va \rrbracket$ it then follows (4), and with (2) we get the desired property $cod_C(\overline{p_0}) \models safe(\overline{p_0})$.

$$va(\overline{p_0}) \wedge \bigwedge_{i=1, \dots, n} asm_i(\overline{p_0}) \models safe(\overline{p_0}). \quad (3)$$

$$cod_C(\overline{p_0}) \wedge \bigwedge_{i=1, \dots, n} asm_i(\overline{p_0}) \models safe(\overline{p_0}) \quad (4)$$

We summarise that if a correct Verification Architecture is given, we only have to show that, firstly, a model is actually a concrete instantiation of the VAs abstract protocol and, secondly, the model fulfils the architecture's assumptions. Then we can conclude the correctness of the entire model.

3 Sequent Calculus for CSP Processes with Data

In this section, we give a short idea of our CSP extension and its embedding into the dynamic logic dCSP that allows for specifying and verifying VAs.

To be able to specify VAs, we need a high degree of freedom to handle general patterns of parametric systems with data. To this end, we introduce an extension to CSP with data constraints to define state changes and a new construct, so-called *undefined processes*. Undefined processes are special processes that allow the occurrence of arbitrary events except for events from a fixed alphabet and arbitrary changes of variables except for variables from a fixed set. Undefined processes can terminate and may be restricted by constraints from an *arbitrary* logic (at least, if this logic has the same semantical domain as CSP with data). On the level of CSP, these constraints are handled as black boxes that restrict the possible behaviour of a process.

Definition 1. *The syntax of CSP processes with data and undefined processes over a set of events $Events$, variables Var , and formulae $Form_\Sigma$ is given by*

$$P ::= \text{Stop} \mid \text{Skip} \mid (a \bullet \varphi) \rightarrow P \mid P_1 \square P_2 \mid P_1 \parallel P_2 \mid P_1 \parallel_A P_2 \mid P_1 \circlearrowleft P_2 \mid X \\ \mid (\text{Proc}_{\setminus A, V} \bullet_{ext} F) \mid (\text{Proc}_{\setminus A, V}^\infty \bullet_{ext} F)$$

where $a \in Events$, $A \subseteq Events$, $V \subseteq Var$, $\varphi \in Form_\Sigma$, and F is a constraint in an external logic *ext*.

In this definition, a difference to the standard CSP definition is that we have constrained occurrences of events $a \bullet \varphi$. As formulae we consider many-sorted first order formulae with predicates and function symbols from a signature $\Sigma = (Sort, Func, Var, Par)$ with primed and unprimed variables Var , parameters Par , and functions $Func$ with sorts from $Sort$. The intuition is that when the event a occurs the state space is changed according to the constraint φ ,

$$\begin{array}{c}
\frac{[a \rightarrow \text{Skip}] \Box \varphi \wedge [a \rightarrow \text{Skip}] [P] \Box \varphi}{[a \rightarrow P] \Box \varphi} \quad (5) \qquad \frac{\psi_{\bar{v}'}^{\bar{v}_0} \Rightarrow \delta_{\bar{v}}^{\bar{v}_0}}{[(a \bullet \psi) \rightarrow \text{Skip}] \delta} \quad (6) \qquad \frac{\psi, F \vdash_{\text{ext}} [\text{Proc} \setminus_{A, V}] \bar{\delta} \quad \bar{\delta} \vdash \delta}{\psi \vdash [\text{Proc} \setminus_{A, V} \bullet_{\text{ext}} F] \delta} \quad (7)
\end{array}$$

Fig. 1. Some example rules from the sequent calculus; the formulae $\psi_{\bar{v}'}^{\bar{v}_0}$ denotes the replacement of a variables \bar{v} in ψ with fresh variables \bar{v}_0 .

where unprimed variables in φ refer to the variable valuations before the occurrence of a and primed variables to the valuations after a . The intuition behind an undefined process like $(\text{Proc} \setminus_{\{a, b\}, \{v\}} \bullet_{DC} F)$ is that during the execution of the process arbitrary behaviour is allowed provided that the DC formula F is not violated. The events a and b are forbidden and the variable v cannot be changed in this execution. An undefined process marked with the ∞ symbol Proc^∞ will never terminate.

We embedded CSP into dynamic logic [9] to reason about CSP processes with data and undefined processes. The idea of this dynamic logic extension dCSP is to use CSP processes with data and undefined processes instead of programs within the box operator $[\cdot]$ and the diamond operator $\langle \cdot \rangle$. The dynamic logic operator $[P] \Box \varphi$ expresses that on all runs of the CSP process always φ holds, whereas $[P] \delta$ states that after every run δ is true. Analogously, $\langle P \rangle \diamond \varphi$ is used to express that there is at least one run where eventually φ holds.

To prove validity of dCSP formulae, we define a set of verification rules in a sequent-style proof calculus. Given finite sets of formulae Δ and Γ , a *sequent* $\Delta \vdash \Gamma$ is an abbreviation for the formula $\bigwedge_{\varphi \in \Delta} \varphi \Rightarrow \bigvee_{\psi \in \Gamma} \psi$. Our sequent calculus consists of rule schemata of the shape $\frac{\Phi_1 \vdash \Psi_1 \quad \dots \quad \Phi_n \vdash \Psi_n}{\Phi \vdash \Psi}$ that can be instantiated with arbitrary contexts, i.e., for every Δ and Γ the rule $\frac{\Delta, \Phi_1 \vdash \Psi_1, \Gamma \quad \dots \quad \Delta, \Phi_n \vdash \Psi_n, \Gamma}{\Delta, \Phi \vdash \Psi, \Gamma}$ is part of the calculus. As usual, formulae above the line are premises and the formula below the line the consequence: if the premises (and possibly some side-conditions) are true then the consequence also holds.

Figure 1 gives some example proof rules. The rule in (5) reduces a CSP prefix expression: to prove that $\Box \varphi$ holds for $a \rightarrow P$ we have to show that *during* execution of a $\Box \varphi$ holds and that *after* the occurrence of a during every run of P also $\Box \varphi$ holds. The following rule (6) reduces a single occurrence of an event a in a process $a \rightarrow \text{Skip}$. The idea is to symbolically execute the data change as defined in the constraint ψ of event a : after an execution of the data change in ψ the post-state of a variable v given by v' need to coincide with the pre-state of this variable in δ . Hence, to show that after every execution of $a \rightarrow \text{Skip}$ the dCSP formula δ holds, we show that the constraint ψ , where every primed variable v' is replaced by a fresh variable v_0 , implies $\delta_{\bar{v}}^{\bar{v}_0}$, i.e., δ , where every v replaced by v_0 . Rule (7) demonstrates how undefined processes with assumptions are handled. To show that on every run of a process $(\text{Proc} \setminus_{A, V} \bullet_{\text{ext}} F)$ the dCSP formula δ is valid, we need to show that a new constraint $\bar{\delta}$ is valid in the logic of F and that in our sequent calculus, $\bar{\delta}$ implies δ . If F is a DC formula then we may show $\bar{\delta}$ with existing proof methods for DC [2]. By this means, our approach flexibly integrates arbitrary timed logics to formulate assumptions on undefined processes.

$$\begin{array}{l}
A = \{check, fail, pass, extend\}, C = \{RD, CT\} \\
System \stackrel{c}{=} FAR \overset{o}{\circ} check \bullet \varphi_{check} \rightarrow \quad \varphi_{check} = \Xi(sf) \wedge sf \leq RD \wedge ok' = false \\
\quad (fail \bullet \varphi_{fail} \rightarrow REC \quad \vee \Xi(sf) \wedge sf > RD \wedge ok' = true \\
\quad \square pass \bullet \varphi_{pass} \rightarrow System) \quad \varphi_{fail} = \Xi(sf) \wedge ok = false \\
\quad \square extend \bullet \varphi_{extend} \rightarrow System \quad \varphi_{pass} = \Xi(sf) \wedge ok = true \\
FAR \stackrel{c}{=} Proc_{\setminus A, C} \bullet F_{FAR} \quad F_{FAR} = \neg \diamond ([sf > RD] \wedge \ell < CT \wedge [sf \leq 0]) \\
REC \stackrel{c}{=} Proc_{\setminus A, C}^{\infty} \bullet F_{REC} \quad F_{REC} = \neg \diamond ([sf > 0] \wedge [sf \leq 0]) \\
\varphi_{extend} = sf' > sf
\end{array}$$

Fig. 2. VA for a small Train Control System

4 Conclusion

Experimental Results. To validate our approach, we verified an architecture for the small example Train Control System in Fig. 2 motivated by the ETCS [1]. We were able to prove the desired safety property $sf > RD \vdash [System] \square sf > 0$ using the presented sequent calculus. To apply rules like rule (7), we made use of automatic DC verification methods [2,10]. In a second step, we proved the correctness of a concrete instantiation of the VA from Fig. 2. This instantiation were given as a COD model, for that direct verification was not possible (timeout after 80h) due to its complexity with 19 real-valued variables, over 300 locations, and 17000 transitions. But as the model is an instantiation of the VA, which can be syntactically checked with a simple refinement rule, we only needed to verify the local DC formulae F_{FAR} and F_{REC} (Fig. 2) to conclude the safety of the entire system. This was done automatically with the PEA toolkit [10] in 7h (F_{FAR}) and 4m (F_{REC}), respectively.

Related work. Our work is inspired by [11], where a fixed DC design pattern for cooperating traffic agents is introduced. Other approaches to combine CSP with data and real-time are, e.g., [7] and [12]. The former, which we also make use of in this work, is not appropriate for a proof-rule base approach because of the more complex combination that integrates CSP, DC, and OZ [13] in an object-oriented class structure. The latter likewise integrates CSP within Z constructs. Further combinations of CSP, OZ, and a real-time language are TCOZ [14] and RT-Z [15]. There is a lot of work in compositional methods for real-time systems: [16,17] introduce a sequent calculus to verify temporal properties for hybrid systems; they also examine fragments of the ETCS as case study. Compositional techniques for the verification of operationally specified real-time systems like timed automata can be found, e.g., in [18,19]. A general view on formalisation techniques for design patterns gives [20], but there, verification of real-time systems is not considered. A related approach using design patterns for a high-level real-time language is [21]: timed automata patterns for a fixed set of timing constraints are given and formally linked to TCOZ.

This is work in progress: We defined CSP with data and undefined processes, the embedding into dynamic logic, and a set of proof rules. Furthermore, the refinement rule for the instantiation of VAs with concrete COD specifications is proven correct. A proof for the correctness of the calculus is not finished yet. We

have tool support [10] for checking local DC assumptions. Tool support for our sequent calculus and the refinement rule is future work. But experiments with examples from the railway domain and automatic verification of the most time-consuming parts (checking DC assumptions) show the success of our method.

References

1. ERTMS User Group, UNISIG: ERTMS/ETCS System requirements specification. <http://www.aeif.org/ccm/default.asp> (2002) Version 2.2.2.
2. Meyer, R., Faber, J., Hoenicke, J., Rybalchenko, A.: Model checking duration calculus: A practical approach. *Formal Aspects of Computing* **20** (2008) 481–505
3. Zhou, C., Hansen, M.R.: *Duration Calculus*. Springer (2004)
4. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall (1985)
5. Roscoe, A.: *Theory and Practice of Concurrency*. Prentice Hall (1998)
6. Woodcock, J., Davies, J.: *Using Z - Specification, Refinement, and Proof*. Prentice Hall, London (1996)
7. Hoenicke, J.: *Combination of Processes, Data, and Time*. PhD thesis, University of Oldenburg, Germany (2006)
8. Faber, J., Jacobs, S., Sofronie-Stokkermans, V.: Verifying CSP-OZ-DC specifications with complex data types and timing parameters. In Davies, J., Gibbons, J., eds.: IFM. Volume 4591 of LNCS., Springer (2007) 233–252
9. Harel, D., Kozen, D., Tiuryn, J.: *Dynamic Logic*. Foundations of Computing. MIT Press (2000)
10. Hoenicke, J., Meyer, R., Faber, J.: PEA Toolkit. <http://csd.informatik.uni-oldenburg.de/projects/epa.html> (2006) University of Oldenburg, Germany.
11. Damm, W., Hungar, H., Olderog, E.R.: Verification of cooperating traffic agents. *International Journal of Control* **79** (2006) 395 – 421
12. Woodcock, J.C.P., Cavalcanti, A.L.C.: A concurrent language for refinement. In Butterfield, A., Pahl, C., eds.: IWF’01. BCS Elec. Works. in Computing (2001)
13. Smith, G.: *The Object Z Specification Language*. Kluwer Academic P. (2000)
14. Mahony, B.P., Dong, J.S.: Blending object-Z and timed CSP: An introduction to TCOZ. In: ICSE. (1998) 95–104
15. Sühl, C.: An overview of the integrated formalism RT-Z. *Formal Asp. Comput* **13** (2002) 94–110
16. Platzer, A., Quesel, J.D.: Logical verification and systematic parametric analysis in train control. In Egerstedt, M., Mishra, B., eds.: HSCC 2008. Volume 4981 of LNCS., Springer (2008) 646–649
17. Platzer, A.: *Differential Dynamic Logics: Automated Theorem Proving for Hybrid Systems*. PhD thesis, University of Oldenburg, Germany (2008)
18. Larsen, K.G., Pettersson, P., Yi, W.: Compositional and symbolic model-checking of real-time systems. In: Proceedings of the 16th IEEE Real-Time Systems Symposium. (1995) 76–89
19. Berendsen, J., Vaandrager, F.W.: Compositional abstraction in real-time model checking. In Cassez, F., Jard, C., eds.: FORMATS. Volume 5215 of LNCS., Springer (2008) 233–249
20. Taibi, T.: *Design Pattern Formalization Techniques*. IGI Publishing, Hershey, PA, USA (2007)
21. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed patterns: TCOZ to timed automata. In Davies, J., Schulte, W., Barnett, M., eds.: ICFEM 2004. Volume 3308 of LNCS., Springer (2004) 483–498

A Formal Model to Develop and Verify Self-Adaptive Systems

Narges Khakpour

Faculty of Electrical and Computer Engineering
Tarbiat Modares University, Tehran
nkhakpour@modares.ac.ir

1 Introduction

Problem Statement Increasingly, software systems are subjected to adaptation at run-time due to changes in the operational environments and user requirements. Adaptation is classified into two broad categories: structural adaptation and behavioral adaptation. While structural adaptation aims to adapt system behavior by changing system's architecture, the behavioral adaptation focuses on modifying the functionalities of the computational entities.

There are several challenges to developing self-adaptive systems. Due to the fact that self-adaptive systems are often complex systems with greater degree of autonomy, it is more difficult to ensure that a self-adaptive system behaves correctly. Hence, one of the main concerns to developing self-adaptive systems is providing mechanisms to trust whether the system is operating correctly where *formal methods* can play a key role. Formal verification of adaptive systems is a young research area [1]. Existing formal methods for analysis of adaptive systems mostly use transition systems and petri nets which are at the low levels of abstraction(see e.g. [2,3,4,5]). So, offering new models to develop self-adaptive systems that provide us formal verification techniques with a high level of abstraction is of a great interest to us.

flexibility is another main concern to achieve adaptation in software systems. Since, hard-coded mechanisms make tuning and adapting of long-run systems complicated, so we need methods for developing adaptive systems that provide a high degree of flexibility. Recently, the use of policies has been given attention as a rich and abstract mechanism to achieve flexibility in the self-managing systems. Although, policies have been used as the adaptation logic for structural adaptation(see e.g. [6,7]) however, fewer work employ policies as a mechanism for behavioral adaptation of self-adaptive software systems. Moreover, structural adaptation has been given strong attention in the research community(see [8]), but fewer approaches tackle behavioral adaptation. Thus, we require new methods to develop systems that provide us behavioral adaptation as well as structural adaptation.

Thesis Statement: The goal of this research is to propose a flexible model, called PobsSAM(Policy-based Self-Addaptive Model) [9], for developing, specifying and verifying self-adaptive systems that uses policies as the principal paradigm to

govern and adapt the system behavior. This model can support both behavioral and structural adaptations. PobsSAM has a formal foundation that employs an integration of algebraic formalisms and actor-based models. The computational (functional) model of PobsSAM is based on actor-based models while an algebraic approach is proposed to specify the policies of manager agents. In this work, we will present verification techniques to analyze PobsSAM models compositionally. PobsSAM can be used as a general model to develop self-adaptive systems. Hence, we will also present an approach to monitor an adaptive software developed by PobsSAM at runtime and verify it against the desired properties.

2 Approach

In this research, we propose a flexible formal model called PobsSAM, to develop, specify and verify self-adaptive systems. PobsSAM uses policies as the main mechanism to adapt and govern system behavior. This section introduces this model in brief.

2.1 PobsSAM Model

A PobsSAM model is composed of a set of Self-Managed Modules(SMMs). An SMM is a set of agents which can manage their behavior autonomously according to the predefined policies. PobsSAM supports interactions of an SMM with the other SMMs in the model. To this aim, each SMM provides well-defined interfaces for interaction with other SMMs. An SMM structure can be conceptualized as the composition of three kinds of entities including:

Managed Actors. Managed Actors are computational actors which are dedicated to the functional behavior of an SMM. The encapsulation of state and computation, and the asynchronous communication make actors a natural way to model distributed systems. Therefore, we use an actor-based model to specify the computational environment of a self-adaptive system. To this aim, an extension of an actor-based language named Rebeca [10] is used.

Views. Views provide a view or an abstraction of the actors state for the managers. A view variable could be the actual state variable, or a function or a predicate applied to the state variables of actors. Views enable managers not to be concerned about the internal behavior of actors and they provide an abstraction of the actor's state for the managers.

Autonomous Managers. Autonomous manager agents are responsible for managing module behavior according to the predefined policies. A manager can operate in different configurations. Each configuration consists of two classes of policies: governing policies and adaptation policies. The managers use governing policies to direct the behavior of actors by sending messages to them. Adaptation policies are used to switch between different configurations. The simple configuration C is defined as $C \stackrel{\text{def}}{=} \langle P, A \rangle$ where P and A indicate the governing policy set and the adaptation policy of C respectively.

2.2 Policy Specification Language

In this section, we present an algebra to specify the governing and adaptation policies.

Governing Policies Whenever a manager receives an event, it identifies all the policies that are triggered by that event. For each of these policies, if the policy condition evaluates to true, its action part is requested to execute by instructing the relevant actors to perform actions through sending asynchronous messages. We express governing policies using a simple algebra as follows, in which P and Q indicate arbitrary policy sets:

$$P, Q \stackrel{\text{def}}{=} P \cup Q \mid P - Q \mid P \cap Q \mid \{p\} \mid \emptyset$$

A simple action policy $p=[o, \varepsilon, \psi, \alpha]$ consists of the priority o , event ε , optional condition ψ and action α . Actions can be composite or simple. A simple action is in form of $r.\ell(v)$ which denotes message $\ell(v)$ is sent to the actor r . Composite actions are created by composing simple actions using sequential composition($\alpha; \beta$), parallel composition($\alpha \parallel \beta$), non-deterministic choice($\alpha + \beta$) and conditional choice ($[\omega? \alpha : \beta]$) operators as follows:

$$\alpha, \beta \stackrel{\text{def}}{=} \alpha; \beta \mid \alpha \parallel \beta \mid \alpha + \beta \mid [\omega? \alpha : \beta] \mid r.\ell(v)$$

Adaptation Policies Whenever an event requiring adaptation occurs, relevant managers in different SMMs are informed. However, adaptation cannot be done immediately and when the system reaches a safe state, the manager switches to the new configuration. Therefore, we introduce a new mode of operation named adaptation mode in which a manager agent runs before switching to the next configuration. There are two kinds of adaptations called loose adaptation and strict adaptation. Under loose adaptation, the manager enforces old policies while in the strict adaptation mode all events will be ignored until the system passes the adaptation mode and reaches a safe state. Adaptation policies are defined using an algebraic language as follows:

$$A \stackrel{\text{def}}{=} [D]_{\delta, \gamma, \lambda, \vartheta} \mid A \oplus A$$

Adaptation policies of a manager is defined as the composition of simple adaptation policies by \oplus operator. Composition of two policies means that those policies are potential to be triggered. The simple adaptation policy $[D]_{\delta, \gamma, \lambda, \vartheta}$, specifies when the triggering condition δ holds and there is no other adaptation policy with the higher priority, the manager evolves to the strict or loose adaptation modes based on the value of λ . When the condition of applying adaptation γ yields, adaptation is performed. D is an arbitrary configuration defined as follows where $[\omega? D : E]$ and $D \square E$ represent conditional and non-deterministic choices respectively and C is a simple configuration:

$$D, E \stackrel{\text{def}}{=} [\omega? D : E] \mid D \square E \mid C \mid 0$$

2.3 Operational Semantics of PobsAM

The operational semantics of PobsAM is defined by the labeled transition systems. There are five classes of deduction rules in PobsAM including (1) the policy enforcement rules which are defined for a manager in the enforcement and loose adaptation modes to enforce governing policies, (2) the policy adaptation rules defined for dynamic adaptation of the manager's policies in the loose and strict adaptation modes, (3) the rules that describe interactions between managers and views, (4) the view rules which reflect the changes of actor's state to the views, and (5) the computation rules which model the functional layer of a PobsAM model.

3 Ongoing Work

Thus far, we proposed a formal model to specify and develop self-adaptive systems. Given the PobsAM model of a system, we can perform different kinds of analysis. We can check the correctness properties of the managers, the actors or both together.

Managers Analysis As stated above, PobsAM has decoupled the adaptation concerns from the functionality of the system. Thus, we can verify the managers independently from the actors provided that we would have a labeled transition system modeling views behavior. This can decrease the complexity of verification task to a great extent. We can perform different types of analysis on the managers which the two important types include:

- *Policy analysis* As policies direct and adapt the behavior of a policy-based self-adaptive system, thus it is required to understand and control the overall effect of policies on the system behavior. In other words, policies often interact with each other that can cause undesirable effects. To this aim, we classify and detect different kinds of conflicts which may exist between policies. We provide different temporal specification patterns to detect conflicts that enable us to automate conflict detection process[11].
- *System stability checking* Adaptation can cause instability in the system, i.e. the adaptation by a manager may lead to another adaptation, which in turn leads to another, and so on. If this cycle continues without reaching a stable state, we say the system is in an unstable state. Verifying the stability of the system is an important property that must be checked. In a PobsAM model, we say a system is unstable when (1) there is a manager waiting for a safe state to switch to the new configuration, but the condition of applying adaptation does not become true and the manager stays waiting in the adaptation mode forever, or (2) there is a policy cycle among the adaptation policies, i.e. applying an adaptation policy causes triggering another adaptation policy and this continues in a cycle. It is noteworthy to mention that all the policy cycles are not unsuitable but some cycles may lead to the oscillation of the system.

Actors Analysis We are aimed at verification of the actors separately from the managers. To this end, we need to model the messages coming from the managers which are the consequence of enforcing governing policies. Rebeca allows us to verify the actors compositionally, however, it assumes that all the coming messages are present in all states. We believe that according to the definition of policies, we can improve the Rebeca compositional verification approach.

Implementation of PobsAM We implemented the managers of a PobsAM model in UPPAAL. Using UPPAAL, we can detect policy conflicts specified by LTL. Moreover, we implemented PobsAM in Maude. Features provided by Maude such as the strategy language and the reflective capabilities make it a suitable option to implement PobsAM.

4 Conclusion and Future Plans

To date, we have presented a flexible modular model for developing and verifying self-adaptive systems which has a formal foundation. PobsAM decouples the adaptation logic of the system from its business logic described at an abstract level using policies. The proposed model permits us to direct/adapt system behavior by enforcing/modifying policies without re-coding actors and managers; thereby it leads to the increase of the system flexibility and scalability. Our future plan includes extending PobsAM to support structural adaptation in addition to compositional and run-time verification of PobsAM models.

Extension with structural adaptation PobsAM can support both behavioral and structural adaptations and our future research will be concentrated on specifying and verifying structural adaptations. Structural adaptation of a PobsAM models is performed by removing/adding an SMM from/to the model or replacing an SMM by another one dynamically. Furthermore, we can allow actors to joint or leave an SMM dynamically. To this end, it is needed to extend adaptation policies to specify structural adaptation as well as behavioral adaptation. This work will be original in using both structural and behavioral adaptations which are directed by an identical mechanism, i.e. adaptation policies.

Compositional verification of the PobsAM models Compositionality is a desirable facility to reduce the complexity of verification process by decomposing a large system into more manageable pieces and proving the correctness of the whole system from that of its immediate components. We can verify a PobsAM model compositionally at four levels: (1) Compositional verification of the managers and actors layers: At present, we can verify the managers independently from the actors. (2) Compositional verification of the SMMs: Regarding the modular nature of the PobsAM models, we plan to present an approach to verify the generic properties of a PobsAM model by checking the properties of an SMM locally and compose the results to prove a property globally. (3) Compositional verification of a manager: A manager can run in different configurations with various goals. Thus, we attempt to verify a formal specification by dividing it

into a set of properties that must be preserved by the configurations. (4) Incremental verification of a manager: Thanks to the modularity of policies, we plan to verify a manager in a specific configuration incrementally.

Run-time analysis Due to the dynamic nature of self-adaptive systems, traditional verification methods applied at the requirement analysis and design stages of development must be enhanced with run-time assurance techniques [1]. Run-time verification is an attractive complement of static analysis. In this technique, software is monitored at run-time and an execution trace of the software is generated. Then, the conformance of that trace is verified against the formal specification. Runtime verification is of special interest to us as it enables us to detect and resolve policy conflicts at runtime. Furthermore, it allows us to adapt the system at run-time by choosing a suitable set of policies, which it may be impossible to be done statically.

References

1. B. H. C. Cheng, H. Giese, P. Inverardi, J. Magee, and R. d. Lemos, "Software Engineering for Self-Adaptive Systems: A Research Road Map", in *Software Engineering for Self-Adaptive Systems 2008*.
2. J. Zhang and B. Cheng, "Model-Based Development of Dynamically Adaptive Software", in *Proceedings of International Conference on Software Engineering, 2006*, pp. 371-380.
3. K. Schneider, T. Schuele, and M. Trapp, "Verifying the adaptation behavior of embedded systems", in *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, Shanghai, China, 2006*, pp. 16 - 22.
4. R. Adler, I. Schaefer, T. Schuele, and E. Vecchie, "From Model-Based Design to Formal Verification of Adaptive Embedded Systems", in *Proceedings of International Conference on Formal Engineering Methods, 2007*, pp. 76-95.
5. S. S. Kulkarni and K. N. Biyani, "Correctness of Component-Based Adaptation", in *Component-Based Software Engineering, 2004*, pp. 48-58.
6. R. J. Anthony and C. Ekelin, "Policy-driven self-management for an automotive middleware", in *Proceedings of 1st International Workshop on Policy-Based Autonomic Computing Florida, USA, 2007*.
7. C. Efstratiou, K. Cheverst, N. Davies, and A. Friday, "An Architecture for the Effective Support of Adaptive Context-Aware Applications", in *Proceedings of the 2nd International Conference in Mobile Data Management (MDM'01), Hong Kong, 2001*, pp. 15-26.
8. J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A survey of self-management in dynamic software architecture specifications", in *Proceedings of the International Workshop on Self-Manages Systems, Newport Beach, USA, 2004*.
9. N. Khakpour, S. Jalili, C. Talcott, M. Sirjani, and M. Mousavi, "PobSAM: Policy-based Managing of Actors in Self-Adaptive Systems," In *Proceedings of the 6th International Workshop on Formal Aspects of Component Software, Eindhoven, The Netherland, 2009*.
10. M. Sirjani, A. Movaghar, A. Shali, and F. S. d. Boer, "Modeling and Verification of Reactive Systems using Rebeca", *Fundamenta Informaticae*, vol. 63, pp. 385-410, 2004.
11. N. Khakpour, R. Khosravi, M. Sirjani, and S. Jalili, "Formal Analysis of Policy-based self-Adaptive Systems", Submitted, 2009.

Exploiting Loop Transformations for the Protection of Software

Enrico Visentini

Università degli Studi di Verona (Italy)
`enrico.visentini@univr.it`

Abstract. Software retains most of the know-how required for its development. Because nowadays software can be easily cloned and spread worldwide, the risk of intellectual property infringement on a global scale is high. One of the most viable solutions to this problem is to endow software with a watermark. Good watermarks are required not only to state unambiguously the owner of a software, but also to be robust and pervasive. In this work we base robustness and pervasiveness on trace semantics. We point out loops as pervasive programming constructs and we introduce loop transformations as the basic block of pervasive watermarking schemes. We survey several loop transformations and classify them into three categories, according to their underlying principles. Then we exploit these principles to build some pervasive watermarking techniques. Robustness still remains a big and challenging open issue.

High quality software is the result of an intellectual effort. Plenty of users can benefit from the result, but only if some producers make the effort. Producers – not users – are entitled to choose a business model for software. Unfortunately software products, or *programs* for short, have some features that issue several challenges to this assumption.

1. It is extremely easy to clone programs and make thousands of illegal copies out of one legally purchased program.
2. Because of a more and more interconnected global network, it is easier and easier to exchange copies of a program.
3. Programs – unlike manufactured goods – mostly retain in themselves the know-how required for their development.
4. Inspecting the content of a program can be simple, especially if the program is expressed in widespread programming languages, like Java or .NET.
5. It is easy to transform a program and embed (part of) it into another program.

It is well understood that the interaction of these five points opens the door to piracy and copyright infringement on a global scale. Yet it is not clear how to tackle them. The first two points seem to be connected to the technological infrastructure we exploit nowadays for storing and exchanging programs. The third point is inherent to the very nature of software. The last point stems from

the fourth one, assuming that one cannot take advantage of what they cannot inspect. Therefore, to thwart piracy, we must try to make inspection unfeasible.

Encrypted programs are difficult, if not impossible, to inspect. But they can be executed only in decrypted form. At runtime encryption thereby fails in protecting programs. Conversely, obfuscated programs do not stop being executable. In fact, *obfuscation* makes the control flow or the data structures of a program harder to analyze, but it preserves both functionality and executability [3]. So far, however, very few provable secure obfuscation schemes are known [10]. If a copyright infringement is likely to take place, *watermarking* can be of help in its detection. A watermark is additional information one embeds in a program to prove ownership [2]. A comment with a copyright notice is a first example of watermark. Unfortunately this kind of watermarks, being totally taken apart from functionality, are easy to distort or remove. A strong connection should exist between a watermark and the functionality of a program, so that you cannot damage one while preserving the other.

We can use sets of traces to detail the functionality of a program [4]. A program P is a set of commands. Each command in P is noted $\ell A_{\ell'}$, where ℓ is a label, A is an action and ℓ' is the label of the command to be executed next [5]. An action consists in testing or changing the value of some variables. As such actual values are stored in environments, commands provide a way to move from one environment to another. If ${}_p A_q$ is a command and ρ is an environment, then $\langle \rho, {}_p A_q \rangle$ is a *state*. By writing two states $\langle \rho, {}_p A_q \rangle \langle \rho', {}_r A'_s \rangle$ in sequence, we assert that A turns ρ into new environment ρ' and $r = q$. A *trace* is a sequence of states. Only so-called initial states can appear at the beginning of a trace. The semantics of a program consists of all the traces starting with an initial state.

The fact that comments play no role in the determination of a trace accounts for the weakness of comments-based watermarks. Good watermarks should be pervasive. We claim here that the more a watermark is pervasive, the more should be the states that are affected by the watermark embedding. Furthermore, good watermarks should be robust. In our view, robustness is the inability of deriving the original semantics from the watermarked program. We claim that there is robustness if the watermark embedding entails a loss of information. Such information is called the *key*. Because of the loss, when you try to characterize the functionality of the watermarked program, you no longer get its semantics \mathcal{S} , but an over-approximation of its semantics, i.e., a set of traces $\mathcal{T} \supset \mathcal{S}$. Traces in $\mathcal{T} \setminus \mathcal{S}$ are misleading, since they describe behaviors and properties that the watermarked program actually has not. For this reason, they thwart the recognition of the original functionality and, consequently, the detection of the watermark. The information that tells good traces from misleading ones is carried by the key. Also, the key enables the detection of the watermark within the good traces. Of course, the key should be held only by the producer of the program. If one tries to distort the watermark or remove it without knowing the key, they should obtain a new program whose functionality is different from the functionality of the original program.

1 For-loops

A good pervasive watermarking scheme is the dynamic path-based one [1], which takes advantage of branching constructs. Branching not only is ubiquitous in non-trivial programs, but it strongly affects evolution of traces. Also looping constructs can be of interest. A looping construct, or *loop* for short, consists of a boolean condition, called the guard, and a set of commands, called the body. The execution of a loop is arranged into iterations. An iteration consists of two steps: first the guard is established to be satisfied; then the body is executed once. The loop stops iterating as soon as the guard is not satisfied. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code [8]. This critical 10% of the code frequently consists of loops [7]. We thereby expect 90% of the states of a trace to be determined by commands involved in loops. On top of that, we also expect loop transformations to be the basic blocks for the design of pervasive watermarking schemes.

In the field of compilers, loop transformations especially target for-loops. A *for-loop* maintains a variable, called the index variable, that has a fixed initialization, and increases by a fixed amount only at the end of each iteration. The for-loop stops as soon as the variable goes beyond a fixed boundary. Formally, a for-loop is a set of commands $P = \{ {}_a\mathbf{F}_b, {}_b\mathbf{G}_c, {}_c\mathbf{H}_d, {}_d\mathbf{I}_b, {}_b\mathbf{G}_e \}$, where \mathbf{G} is the guard testing the index variable against the boundary, \mathbf{H} is the body, and \mathbf{F} and \mathbf{I} respectively initializes and updates the index variable. An iteration of P is a trace of the form $\langle \rho_1, {}_b\mathbf{G}_c \rangle \langle \rho_2, {}_c\mathbf{H}_d \rangle \langle \rho_3, {}_d\mathbf{I}_b \rangle$, where ρ_i with $i \in \mathbb{N}$ are environments and ${}_b\mathbf{G}_c$ accounts for the satisfied guard. A full execution of P is a trace σ consisting of a state with command ${}_a\mathbf{F}_b$, a sequence of iterations and finally a state with command ${}_b\mathbf{G}_e$ accounting for the unsatisfied guard:

$$\begin{aligned} & \langle \rho_0, {}_a\mathbf{F}_b \rangle \langle \rho_1, {}_b\mathbf{G}_c \rangle \langle \rho_2, {}_c\mathbf{H}_d \rangle \langle \rho_3, {}_d\mathbf{I}_b \rangle \\ & \quad \langle \rho_4, {}_b\mathbf{G}_c \rangle \langle \rho_5, {}_c\mathbf{H}_d \rangle \langle \rho_6, {}_d\mathbf{I}_b \rangle \\ & \quad \dots \\ & \quad \langle \rho_7, {}_b\mathbf{G}_c \rangle \langle \rho_8, {}_c\mathbf{H}_d \rangle \langle \rho_9, {}_d\mathbf{I}_b \rangle \langle \rho_{10}, {}_b\mathbf{G}_e \rangle . \end{aligned} \tag{1}$$

The semantics of P is the set of all traces whose first state has command ${}_a\mathbf{F}_b$. It is straightforward to notice that by collecting all the commands along a trace we get a program $P' \subseteq P$ [5]. In particular, each iteration is abstracted to $\{ {}_b\mathbf{G}_c, {}_c\mathbf{H}_d, {}_d\mathbf{I}_b \}$; hence all iterations *collapse* to the same set of commands.

Each value assumed by the index variable identifies an iteration of the for-loop. The set I of all such values is called *iteration space* and can be naturally represented on a line. We have a *loop nest* if the body of a for-loop is itself a for-loop. In such case we have two index variables and we identify iterations by using two values, collected in two-dimensional *index vectors*. Here iteration space I is the set of all index vectors and can be graphically represented as a two-dimensional polyhedron. In I an ordering is defined that reflects the order in which the iterations of the loop nest take place. We say that such ordering describes a *traversal* of I .

2 A Taxonomy for Loop Transformations

The notions of iteration space and traversal account for the ease of predicting the behavior of a for-loop. This predictability is helpful in the design of transformations, because it usually makes clear which requirements and consequences a transformation has. It is no surprise, therefore, that more than ten for-loop transformations are known [9]. We propose to classify them into three categories.

The first category includes transformations that perform a remapping of the iteration space while preserving the total number of iterations. The simplest member is perhaps *loop reversal*, that inverts the order in which the iterations of the for-loop are performed. In the polyhedron representing the iteration space, this corresponds to flip the traversal by applying a reflection with respect to the center of the polyhedron. Among the transformations of this category, there are also *loop interchange*, *loop tiling* and *loop skewing*.

The second category includes a pair of loop transformations that preserve iteration spaces, but perturb bodies. In particular, *loop fission* takes two loops with the same iteration space and derives a new loop by merging their bodies together; *loop distribution* goes the other way round:

$$\left\{ \begin{array}{l} \mathbf{aF}_b, \mathbf{bG}_c, \mathbf{cH}_d, \mathbf{dI}_b, \mathbf{bG}_p \\ \mathbf{pF}_q, \mathbf{qG}_r, \mathbf{rH}'_s, \mathbf{sI}_q, \mathbf{qG}_z \end{array} \right\} \begin{array}{c} \xrightarrow{\text{fission}} \\ \xleftarrow{\text{distribution}} \end{array} \{ \mathbf{aF}_b, \mathbf{bG}_c, \mathbf{cH}_r, \mathbf{rH}'_d, \mathbf{dI}_b, \mathbf{bG}_z \}$$

The third category includes transformations that, by changing the labels of some commands, partition the set of iterations into classes and allow only iterations in the same class to collapse together. Consider for instance *loop peeling*, which splits the first iteration out of the for-loop. Peeling turns (1) into:

$$\begin{aligned} &\langle \rho_0, \mathbf{aF}_p \rangle \langle \rho_1, \mathbf{pG}_q \rangle \langle \rho_2, \mathbf{qH}_r \rangle \langle \rho_3, \mathbf{rI}_b \rangle \\ &\quad \langle \rho_4, \mathbf{bG}_c \rangle \langle \rho_5, \mathbf{cH}_d \rangle \langle \rho_6, \mathbf{dI}_b \rangle \\ &\quad \dots \\ &\quad \langle \rho_7, \mathbf{bG}_c \rangle \langle \rho_8, \mathbf{cH}_d \rangle \langle \rho_9, \mathbf{dI}_b \rangle \langle \rho_{10}, \mathbf{bG}_e \rangle . \end{aligned} \tag{2}$$

As a consequence, the iterations are partitioned into two classes: one class with the first iteration only and one class with all the others. If we abstract trace (2), we obtain $\{ \mathbf{aF}_p, \mathbf{pG}_q, \mathbf{qH}_r, \mathbf{rI}_b, \mathbf{bG}_c, \mathbf{cH}_d, \mathbf{dI}_b, \mathbf{bG}_e \}$, where \mathbf{pG}_q , \mathbf{qH}_r and \mathbf{rI}_b account for the first class of collapsing iterations and \mathbf{bG}_c , \mathbf{cH}_d and \mathbf{dI}_b account for the second class. A more sophisticated collapsing pattern is provided by *loop unrolling* which, by executing pairs of iterations sequentially, halves the total number of iterations performed. As shown in [6], unrolling transforms (1) into:

$$\begin{aligned} &\langle \rho_0, \mathbf{aF}_b \rangle \langle \rho_1, \mathbf{bG}_c \rangle \langle \rho_2, \mathbf{cH}_d \rangle \langle \rho_3, \mathbf{dI}_{b'} \rangle \\ &\quad \langle \rho_4, \mathbf{b'G}_{c'} \rangle \langle \rho_5, \mathbf{c'H}_{d'} \rangle \langle \rho_6, \mathbf{d'I}_b \rangle \\ &\quad \dots \\ &\quad \langle \rho_7, \mathbf{b'G}_{c'} \rangle \langle \rho_8, \mathbf{c'H}_{d'} \rangle \langle \rho_9, \mathbf{d'I}_b \rangle \langle \rho_{10}, \mathbf{bG}_e \rangle . \end{aligned} \tag{3}$$

Again, we have two collapsing classes: one collecting iterations with odd index vector, one collecting the others. By abstraction, we get unrolled for-loop $\{ {}_a\mathbf{F}_b, {}_b\mathbf{G}_c, {}_c\mathbf{H}_d, {}_d\mathbf{I}_{b'}, {}_{b'}\mathbf{G}_{c'}, {}_{c'}\mathbf{H}_{d'}, {}_{d'}\mathbf{I}_b, {}_b\mathbf{G}_e \}$, where ${}_b\mathbf{G}_c$, ${}_c\mathbf{H}_d$ and ${}_d\mathbf{I}_{b'}$ account for the first class of collapsing iterations and ${}_{b'}\mathbf{G}_{c'}$, ${}_{c'}\mathbf{H}_{d'}$ and ${}_{d'}\mathbf{I}_b$ account for the second class. The body of the resulting loop is set of commands $\{ {}_c\mathbf{H}_d, {}_d\mathbf{I}_{b'}, {}_{b'}\mathbf{G}_{c'}, {}_{c'}\mathbf{H}_{d'} \}$. Other members of the this category are *index splitting* and *index merging*.

3 Watermarking Schemes

The aim of our taxonomy is to point out the principles of loop transformations, in view of their exploitation for sake of watermarking.

Consider for instance the principle of collapsing iterations. Before undergoing a transformation t of the third category, a for-loop is a set of commands that accounts *implicitly* for a whole set of iterations. From the previous section we know that t partitions the set of iterations into several collapsing classes. For each class, commands are provided that *implicitly* account for all and only the iterations of that class. We notice however that such commands *explicitly* account for the class itself, thus partially preventing the loss of information that takes place before the transformation is applied, i.e., when all iterations collapse together. It is clear that the more refined is the partition, the more information is made explicit. Therefore t can be said to act as an extractor of information and could be exploited to extract watermarks. In [6] for instance we propose a watermarking scheme based on loop unrolling. The watermark is a value that is held in a variable. The correct computation of such value starts at an odd iteration and finishes at the next one, which is even. From the previous section we know that loop unrolling is a transformation that explicits the class of odd iterations and the class of even iterations. So the watermark can be easily retrieved by analyzing the unrolled loop.

Another idea, still to be explored, is to use the loop transformations themselves to encode a watermark. Let P be a program with several for-loops and $\sigma = 1101$ be a binary string representing the watermark to be embedded in P . Let t_0 and t_1 be two loop transformations. We define the watermarked program to be a program P' such that $t_1(t_0(t_1(t_1(P')))) = P$. For sake of reliability, we make the following requirements:

- each time a t_i with $i \in \{0, 1\}$ is applied, there is only one for-loop that can be the target of t_i ;
- none of t_0 and t_1 is an involution or an idempotent operator;
- the composition of t_0 and t_1 is not commutative.

It seems appropriate to choose t_0 from the first category and t_1 from the second category, so that they can affect distinct targets (t_0 affects iteration spaces, while t_1 affects bodies). Moreover, both t_0 and t_1 are legal as long as they do not create or disrupt dependencies between iterations [9]. Thus we could control the order of application of both the transformations by introducing and removing fake dependencies.

It is not clear yet how the proposed schemes can be made robust. A major drawback is that loop transformations were introduced to improve the performance of loops [9]. They were not thought to be robust; indeed, because of the benefits they can grant, there is usually no point in undoing them. As a part of a watermarking scheme, however, they are aimed in the first place at the robust embedding of information. This is likely to jeopardize performance. We thereby need to assess the performance penalty induced by our schemes. On the other side, robustness might come from a detailed comprehension of what loop transformations do, how they cooperate and on which conditions they can be undone. This is especially important since loop transformations themselves may be used to distort watermarks and obstruct their correct extraction. A canonical form of for-loop may be useful to tackle this problem.

References

1. C. Collberg, E. Carter, S. Debray, A. Huntwork, J. Kececioglu, C. Linn, and M. Stepp. Dynamic path-based software watermarking. *SIGPLAN Not.*, 39(6):107–118, 2004.
2. C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. Technical Report TR00-03, University of Arizona, Feb, 10 2000.
3. C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, University of Auckland, July 1997.
4. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
5. P. Cousot and R. Cousot. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Conference Record of the 19th ACM Symposium on Principles of Programming Languages*, pages 178–190, New York, 2002. ACM Press.
6. M. Dalla Preda, R. Giacobazzi, and E. Visentini. Hiding software watermarks in loop structures. In Maria Alpuente and German Vidal, editors, *Static Analysis: Proceedings of the 15th International Static Analysis Symposium*, volume 5079, pages 174–188. Springer, July 2008.
7. J. W. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Technical report, Proc. Compiler Construction '96, 1995.
8. J. L. Hennessy, D. A. Patterson, and A. C. Arpaci-Dusseau. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.
9. S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, second edition, August 1997.
10. C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 12 2000.

Rigorous Development of Java Card Applications With the B Method

Bruno Gomes¹

Advisors: David Déharbe¹ and Anamaria M. Moreira¹

Federal University of Rio Grande do Norte; Natal, RN; Brazil
`bruno@consiste.dimap.ufrn.br`

Abstract. Smart Card applications usually require reliability and security to avoid incorrect operation or access violation in transactions and corruption or undue access to stored information. A way to reach these requirements is improving the quality of the development process of these applications. BSmart is a method and a corresponding tool to support the formal development of Java Card smart card applications, following the B formal method, the application being translated from a concrete refinement of its formal B abstract specification.

1 Introduction

A smart card is a resource constrained computer device able to store data and to run small-sized software in a secure way. The smart card system is composed by the card-side “server” application (the applet) and by a client application (the host application) outside the card. These applications communicate exchanging packets of data in a low level protocol, the host sending a request to an applet’s service, which in turn sends back a response containing the results of the service processing.

Smart card applications are present in our everyday life in a wide range of sectors such as banking and finance, communication, Internet, public transport, health care, etc. These applications usually manage confidential information, such as bank account data, the medical history of a patient or user authentication data. Thus, to prevent undesirable behavior and to avoid security violations it is helpful to adopt a rigorous software engineering process, methods and tools during its development to ensure a final product in conformance with the specified requirements.

Java Card [1] is a version of the Java platform with a restricted API and Virtual Machine optimized for smart cards and other memory and processor constrained devices. Some Java features, such as dynamic class loading, threads, *strings*, the types *float* and *double*, and multi-dimensional arrays are not present in the current version of Java Card. However, the Java Card developer can benefit from most of the Java features, such as portability, type-safe language, object oriented development, and a wide range of available tools.

This PhD thesis proposes a method and its corresponding tool support, named BSmart, which aims to improve the quality of the Java Card smart card

applications through a development process that automates the generation of the card-side Java Card application and a Java API for the host, based on the refinement of its formal specification written in B [2] notation.

It is important to note that the B formalism has been chosen because of its adequacy for the specification of API software, which is the core of a Java Card application. B also has a well established tool support and successful histories of adoption in industry scale [3] and academic [4] [5] projects.

Previous work [6] [7] introduced the foundations of the *BSmart* method. This paper describes the general ideas of the method and the actual state of its tool support in Sections 2 and 3. Some open issues to be tackled during the PhD thesis are listed in Section 4 and final considerations are presented in Section 5.

2 The BSmart Method

The BSmart method aims to abstract the details of the Java Card smart card system for the specifier as much as possible. Most of his/her work is focused in specifying the functionality of the applet without the need to take into account the specific aspects of the Java Card low level communication protocol as well as the treatment of data processing (coding/decoding) between the host and card applications.

In order to translate the B specification to its corresponding Java Card application code, the method requires some refinement steps that progressively add the requirements of the Java Card application framework, as explained in the next subsection. As usual in the B method, other user defined refinements may be needed to take the specification translatable to Java Card. All of these developed modules, before the translation phase, must be verified using B tools, which include a type checker, a proof obligation generator and provers. The success of the verification ensures the consistency of the specified properties and also that the refinements are in accordance with the abstract properties.

An overview of the development process following the method is shown in Figure 2. In this process, the specifier starts from an initial high level and non-Java Card related B specification of the application (*APP.mch*, in the figure). Two sequences of refinement/implementation of this specification are then applied, one for the card-side and other for the host-side development.

2.1 Card-side development

To develop the smart card implementation of the API we require at least a refinement (*APP_FF.ref*), usually applied to make it closer to a Java Card code. We achieve this making it full-function, i.e., weakening the preconditions of the operations so that they only define typing of the parameters. The remaining restrictive conditions are handled in its body through conditional substitutions, whose non-validity leads to the throwing of an exception. This is performed by modeling simple Java Card exception classes, such as *ISOException*. The identifier code of each exception and any other constants or Java Card related

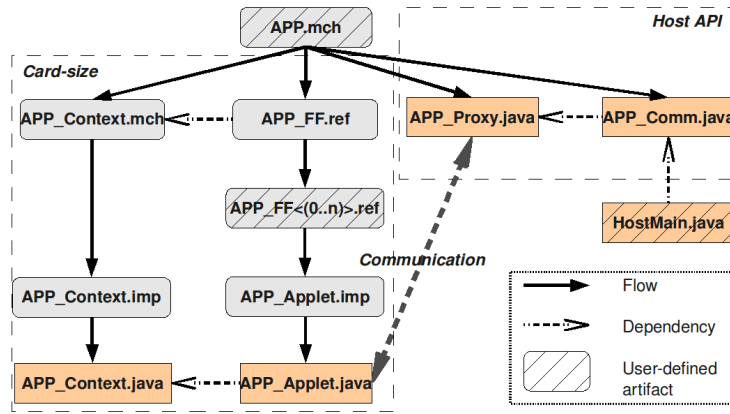


Fig. 1. A view of the BSmart method development and its artifacts

information the refinement needs are included in a separated context machine, named *APP_Context.mch*.

Other refinements may be required, for example, to adapt the types of the original specification to Java Card compatible types. This is the case when a client host application sends data using the Java *int* type, whose implementation is not mandatory in Java Card. Thus, one must represent an *int* as two *short*'s or as an array of *bytes* to avoid loss of precision. Corresponding machines that model these types are included as part of a library of reusable machines in the tool support for the method (Section 3). The automation of the adaptation was not yet added in the tool, but an approach to do this within the strict rules of B refinement is described in [8].

The translation of the B implementations into Java Card code is the last stage in the card-side development. The implementation is a special kind of refinement containing only B constructs, in a subset of the notation called B0, directly translatable to the target language. The main B0 implementation is that for the applet, containing the services offered to the host, but other modules may have to be generated, such as the implementation for the context machine.

2.2 Host-side development

The second line of development takes care of the implementation of the API on the host side and is fully automatable. The generation of the API components is performed from the original specification (*APP.mch*) since we only need to know the expected services of the applet and the necessary data to process them, information that can be obtained from the signature of each *APP.mch* operation.

For the host developer are generated two main classes, named *APP_Comm* and *APP_Proxy*, the former is seen by the user host application and contains high-level Java methods to call each applet service and to control the life-time

of the applet. The service calls are dispatched to the *Proxy* class, which is responsible for coding the received data into a packet in the Java Card protocol format and send it to the applet. This class also decodes the returned response, sending it back to the to *APP_Comm* class.

Using the generated API the user does not need to directly manipulate the lower-level specificities of the Java Card protocol, increasing this way its productivity and allowing us to freely change the API for communication and the internal algorithms for data encoding/decoding without impact to the front-end user developed host application (*HostMain.java*).

3 Tool Support for the BSmart Method

The BSmart tool [9] is an Eclipse plugin connecting several software components, each responsible for implementing a different step of the BSmart method. Essential software for the B formal method is also included, such as a type checker, and connection with external tools, such as Atelier B, for proof obligation generation and verification. As noted before, we supply jointly with the tool a library of B modules modeling essential classes of the Java Card API, types and useful data structures, including its corresponding Java Card implementations. The development of the library is still in progress and includes some prototypes that can be reused by specifiers and Java Card developers with the advantage of being fully verified using the B method.

The main components that provide support for the method are the *BSmart Modules Generator* and the *B to Java Card code translator*. The former is responsible for generating the context machine and the full-function refinement required by the method and the latter translates all the B implementation modules into Java Card programming code and also generates the API classes for the host side client. The translator has been developed based on the Java translator of JBtools [10].

4 Open Issues

In the following, we describe some open issues in the actual stage of the PhD research that are interesting to be taken into consideration.

Verification of the translation The transformations involved in the automatic generation of B required refinements are subject to correctness verification through the use of B tools. These tools can check, for instance, syntax, typing, invariant properties and refinement invariant and constraints. However, a critical verification that is not yet done is the verification of the translation of the B implementation into the Java Card application. An approach could be the translation of JML assertions in the generated code, as in the work of [11]

Development of a real world case study We have been using only small samples of smart card applications to test the method and its corresponding tool support, as an application for public transportation payment. To better validate the method and the tool, we need a more robust real world case study. A strong candidate is the specification in B of the Mondex electronic purse, a case study that is part of the Verified Software Initiative. In the Mondex system some amount of monetary value is transferred from a source to a target smart card purse in a non-atomic protocol. Each purse must be implemented in isolation, without sharing properties through a global control. The Mondex system has been formally specified in Event-B in Butler and Yadav [4] work and in other work using several formalisms. We started to adapt this system specification to a programming specification, extracting the card and host specifications, modeling them according to the BSmart method.

Expansion of the tool library We have developed B machines to model some classes of the Java Card API, Java and Java Card primitive types. We plan to supply these verified B models to all classes of the Java Card API and to other useful tasks for Java Card applications, such as manipulation of time and data and currency conversions. Corresponding classes generated from these modules can be reused by Java Card developers, this way contributing to the correctness of the application as a whole. For the Java Card API specification, the work of Meijer [5] and Larsson [12] could be a starting point, as well as its official documentation.

Object oriented translation The B notation for the implementation module has some constructions similar to an imperative language like C. For that reason, the industrial tools for B, such as the Atelier B and the B-Toolkit, provide translators to this language paradigm. The Rodin platform plugin UML-B [13], presents solutions to overcome the B object orientation limitations, when translating from UML class diagrams into B. In our simple approach, a B implementation and sets are directly translated to a Java Card class. However, in some cases, a non automated fine control of some aspects of the translation may be required, for example, to control the access level of a variable or to apply modifiers to a class or method. In that cases, we could complement the B source with some annotations or configuration files to help the translator.

Optimization of the translation Since the requirements of memory storage and processor power of a smart card are often critical, it is ideal to generate the card-side code as optimized as possible. Some optimizations could be to allocate all the memory an applet needs to persist in its constructor, to reduce, when possible, the number of local variables, etc. Requet e Bossu [14] and Bert et. al [15] work present a similar problem, addressing the translation from B specifications to C programs with optimizations in the context of the BOM (B with Optimized Memory) project.

5 Conclusions

The BSmart method allows the development of a Java Card application in a model driven development (MDD) process following the B method. The work has started as a master degree research. We started to identify the common aspects of a Java Card application, and to develop B specifications of some simple applications. These case studies evolved to the BSmart method and the first version of its tool support.

We can say that the development process of the master research is more systematic than rigorous. The doctoral work (currently in the second year) aims to improve this results with a more robust approach. We are interested in prioritize the verification process of the B to Java/Java Card translations (Host API and card-side classes), and in exploring more sophisticated techniques to generate the B models and the Java Card application final code, possibly using some MDD tool, such as TXL [16]. Currently, this process is performed using only internally developed algorithms. The use of other formalisms, such as JML for runtime verification, is also under investigation. Finally, to better validate the proposal and its tool support, we are studying how to adapt the Mondex case study to the development with B, following the BSmart method.

References

1. Chen, Z.: Java Card Technology for Smart Cards: Architecture and Programmer's Guide. Addison Wesley (2000)
2. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge (1996)
3. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Météor: A successful application of B in a large project. In: Formal Methods 99. Volume 1708 of LNCS. (1999)
4. Butler, M., Yadav, D.: An Incremental Development of the Mondex System in Event-B. Formal Aspects of Computing **20**(1) (2007) 61–77
5. Meijer, H., Poll, E.: Towards a full formal specification of the Java Card API. In: Smart Card Prog. and Security. Number 2140 in LNCS, Springer (2001) 165–178
6. Gomes, B., Moreira, A.M., Déharbe, D.: Developing Java Card applications with B. In: Brazilian Symposium on Formal Methods (SBMF). (2005) 63–77
7. Deharbe, D., Gomes, B.G., Moreira, A.M.: Automation of Java Card component development using the B method. In: ICECCS, IEEE Comp. Soc. (2006) 259–268
8. Déharbe, D., Gomes, B.G., Moreira, A.M.: Refining Interfaces: The Case of the B Method. Technical report, Fed. Univ. of Rio Grande do Norte (2009) .To appear.
9. Déharbe, D., Gomes, B.G., Moreira, A.M.: Bsmart: A Tool for the Development of Java Card Applications with the B Method. In: ABZ. (2008) 351–352
10. Voisinet, J.C.: JBtools: an experimental platform for the formal B method. In: Principles and Practice of Programming (PPPJ'02), Dublin, ACM (2002) 137–139
11. Costa, U., Moreira, A., Musicante, M., Neto, P.: Specification and runtime verification of java card programs. In: Brazilian Symp. on Formal Methods. (2008)
12. Larsson, D.: OCL specifications for the Java Card API. Master's thesis, School of Comp. Science and Engineering, Göteborg University (2003)
13. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by uml. ACM Trans. Softw. Eng. Methodol. **15**(1) (2006) 92–122

14. Requet, A., Bossu, G.: Embedded formally proved code in a smart card: Converting B to C. In: 3rd ICFEM, York, UK, IEEE Computer Society (2000) 15–
15. Bert, D., et al.: Adaptable translator of B specifications to embedded C programs. In: FME. Volume 2805 of LNCS. (sept. 2003) 94–113
16. Cordy, J.: The TXL Programming Language (2009) Available on: <http://www.txl.ca>. Accessed on: jul. 2009.

Towards a Theory for Timed Symbolic Testing

Sabrina von Styp-Rekowski

RWTH Aachen University, Germany
sabrina.von-styp@rwth-aachen.de

Abstract. This paper introduces an implementation relation $\mathbf{stioco}_{\mathcal{D}}$ for symbolic timed automata, which generalises the well-known \mathbf{ioco} implementation relation to systems with data and real-time behaviour. We choose a purely symbolic approach to define this relation, which is based on similar work of Frantzen et al. for Symbolic Transition Systems.

1 Introduction

Formal methods have been applied successfully in the field of conformance testing over the last decade. A well-known representative is the \mathbf{ioco} framework [8], where the correct behaviour of an implementation under test is specified using a formal model (a labelled transition system (LTS) with input and output actions), and which allows for the automatic derivation of test-suites from this specification. Test-cases cast their verdicts following a formal correctness criterion: the implementation relation \mathbf{ioco} . The \mathbf{ioco} relation describes unambiguously when an implementation (which, due to the testing-hypothesis, must be representable as an LTS) conforms to a specification.

The \mathbf{ioco} theory has been extended in several directions. Recently, Frantzen et al. [5] have developed an approach to deal with data input and output to and from the implementation, respectively. Data is treated symbolically, which avoids the usual problems of state-space explosion and infinite branching that occur when data is represented explicitly in an ordinary LTS . Specification and implementation are modelled here as Symbolic Transition Systems (STS), which are LTS extended with a notion of data and data-dependent control flow based on first order logic. In the context of this theory, the implementation relation \mathbf{stioco} has been developed, which is defined solely within the FO-Logic framework on STS level.

Also several approaches to timed \mathbf{ioco} -based testing have been developed [6,3,2,4], mostly on the basis of timed automata. Different notions of conformance have been defined on the basis of timed $LTS(TLTS)$. A detailed comparison of the different notions of conformance is given by Schmaltz et al. [7].

In this paper we take first steps towards a testing theory which combines time and data. Our main contribution is the definition of a conformance relation $\mathbf{stioco}_{\mathcal{D}}$ for *Symbolic Timed Automata* (STA). STA are a straightforward combination of STS and TA , but which allow data inputs to influence the real-time behaviour. The semantics of STA is given in terms of $TLTS$. $\mathbf{stioco}_{\mathcal{D}}$ is an

extension of **sioco** for *STA*, and can be shown to coincide with **tioco** on the semantical, *TLTS* level.

Our approach is to transform an *STA* into an *STS*-like formalism, by encoding the timing aspects within the FO-framework. The resulting structure is called Delayed Symbolic Transition Systems (*DSTS*). We then follow the approach of Frantzen [5] with some essential modifications to define **stioco** _{\mathcal{D}} .

Overview. In Section 2 we introduce *STA* and their semantics. In Section 3 we describe the translation of *STA* to *DSTS*, and the *DSTS* semantics in terms of *TLTS*. Afterwards, in Section 4 we define an implementation relation **stioco** _{\mathcal{D}} for *DSTS*s.

2 Symbolic Timed Automaton

Symbolic timed automata (*STA*) are a combination of Symbolic Transition Systems and Timed Automata. We denote with $\mathcal{B}(\mathcal{C})$ the usual set of clock constraints over a set of clocks \mathcal{C} (*cf.* [1]). The set $\mathcal{B}(\mathcal{C} \cup \mathcal{V})$ is the set of clock-constraints over clock set \mathcal{C} , where upper or lower bounds can be variables from a set of integer variables \mathcal{V} . We assume a certain First-Order-Structure (usually with the an universe of integers), and denote with $\mathfrak{F}(Var)$ the set of FO-Formulas over variable set Var . $\mathfrak{T}(Var)$ denotes a set of terms over Var .

Definition 1 (Symbolic Timed Automata). *A symbolic timed automaton is a tuple $\mathcal{A} = (\mathcal{L}, l_0, \mathcal{V}, \mathcal{J}, \mathcal{G}, \mathcal{C}, Inv, \rightarrow)$ where \mathcal{L} is a set of locations, l_0 is the initial location, \mathcal{C} is a set of clock variables, $Inv : \mathcal{L} \rightarrow \mathcal{B}(\mathcal{C})$ assigns invariants (clock constraints of the form $x < c$ or $x \leq c$) to locations, \mathcal{V} is a set of location variables, \mathcal{J} is a set of interaction variables, \mathcal{G} is a set of action gates, where \mathcal{G}_I is a set of input and \mathcal{G}_U the set of output gates with $\mathcal{G}_I \cap \mathcal{G}_U = \emptyset$, and finally $\rightarrow \subseteq \mathcal{L} \times \mathcal{G}_I \cup \mathcal{G}_U \times \mathfrak{F}(Var) \times \mathfrak{T}(Var)^\mathcal{V} \times \mathcal{B}(\mathcal{C} \cup \mathcal{V}) \times 2^{\mathcal{C}} \times \mathcal{L}$ is a set of switches, where $Var = \mathcal{V} \cup \mathcal{J}$.*

We write $l \xrightarrow{g\langle i \rangle, \varphi, \rho, \theta, \lambda} l'$ for $(l, g\langle i \rangle, \varphi, \rho, \theta, \lambda, l') \in \rightarrow$. $?g\langle i \rangle$ is an input gate, binding incoming data to variable i , and $!g\langle i \rangle$ is an output, sending datum i . A switch can only fire if its *data guard* $\varphi \in \mathfrak{F}(Var)$, and the clock constraint $\theta \in \mathcal{B}(\mathcal{C} \cup \mathcal{V})$ is fulfilled. φ imposes not only constraints on the current valuation of location variables, but also on the values that can be received or sent over the corresponding gate. θ is a clock constraint where bounds on clocks and clock differences can be location variables. This is the only interaction of data and time that we allow. ρ describes the updates of location variables. In particular, interaction variables can be assigned to location variables. λ is a set of clocks which is reset to 0 upon firing the switch. Invariants $Inv(l)$ determine, as in timed automata, the conditions which allow staying in a location.

Note that *STA* without clocks, invariants, and the θ and λ components on switches, are just ordinary *STS*, as defined in [5].

3 DSTS

In this section we introduce Delayed Symbolic Transition Systems (*DSTS*), which help us to reuse Frantzens theory for *STS*s to define **stio** $_{\mathcal{D}}$ (see Figure ??). A *DSTS* is in essence a reformulation of an *STA*, where the real-time aspects are encoded in FO logic. We then adjust Frantzens approach for our purposes, to obtain **stio** $_{\mathcal{D}}$. It can be shown that **stio** $_{\mathcal{D}}$ coincides on *TLTS* level with **tioco**.

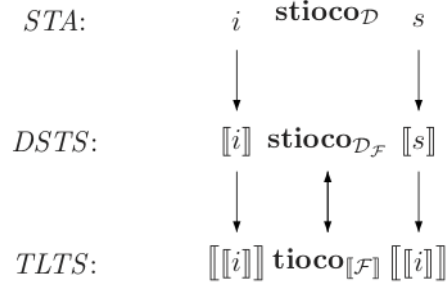


Fig. 1. Semantic hierarchy for *STA*

3.1 Delayed Symbolic Transition Systems

Our approach to define an implementation relation for *STS* is to encode the time-related concepts in FO logic, and obtain thus an *STS*. The resulting delayed *STS* (*DSTS*) is defined over a two-sorted FO-Logic, where \mathfrak{U} denotes the universe for data (usually integers) and \mathfrak{U}_t the time domain (usually reals). The step from *STA* to *DSTS* is straightforward. Assume a switch $l \xrightarrow{g, \varphi, \rho, \theta, \lambda} l'$. Clock constraint θ is translated into an FO-Formula and combined with data constraints φ . Clock reset λ is translated into a term mapping $\lambda_{FO} : \mathcal{C} \rightarrow \mathcal{C} \cup \{0\}$ with $\lambda_{FO}(c) = 0$ if $c \in \lambda$, and $\lambda_{FO}(c) = c$, otherwise. λ_{FO} is then combined with the update mapping ρ . Invariant $Inv(l)$ is transformed into FO-Formula $\kappa \in \mathfrak{F}(\mathcal{C})$. We introduce term mapping $\pi : \mathcal{C} \mapsto \mathfrak{T}(\mathcal{C} \cup \mathcal{T})$ as $\pi(c) \mapsto c + d$, where $d \in \mathcal{T}$ is a *delay variable*. For each location l in the *STA* we introduce two locations \check{l}, \hat{l} in the *DSTS*. The transition $\check{l} \xrightarrow{\delta, \kappa, \pi} \hat{l}$ describes then the sojourn conditions of location l . The firing of the original switch is described by a *DSTS*-switch $\hat{l} \xrightarrow{g, \varphi \wedge \theta_{FO}, \rho \cup \lambda_{FO}} \check{l}'$. More concisely,

$$\frac{l \xrightarrow{g, \varphi, \rho, \theta, \lambda} l'}{\check{l} \xrightarrow{\delta, \kappa, \pi} \hat{l} \xrightarrow{g, \varphi \wedge \theta_{FO}, \rho \cup \lambda_{FO}} \check{l}'}$$

The semantics of an initialised *DSTS* is given by a *TLTS*.

Definition 2 (Semantics). Let $\mathcal{A} = (\mathcal{L}, l_0, \mathcal{V}, \mathcal{J}, \mathcal{G}, \mathcal{C}, Inv, \rightarrow)$ be an STA, and $\mathcal{S} = D(\mathcal{A})$ its corresponding DSTS. Its interpretation $\llbracket \mathcal{S} \rrbracket_\iota$ in the context of an initialisation $\iota \in \mathfrak{U}^{\mathcal{V}}$ and $\zeta_0 = [\mathcal{C} \mapsto 0]$ is defined as the TLTS $\llbracket \mathcal{S} \rrbracket_\iota = (\mathcal{L} \times \mathfrak{U}^{\mathcal{V}} \times \mathfrak{U}_t^{\mathcal{C}}, (l_0, \iota, \zeta_0), \Sigma, \rightarrow)$, where

- $\Sigma = \bigcup_{g \in \mathcal{G}} (\{g\} \times \mathfrak{U}) \cup \bigcup_{\delta \in \mathcal{D}} \mathfrak{U}_t$
- $\rightarrow \subseteq (\mathcal{L} \times \mathfrak{U}^{\mathcal{V}} \times \mathfrak{U}_t^{\mathcal{C}}) \times \Sigma \times (\mathcal{L} \times \mathfrak{U}^{\mathcal{V}} \times \mathfrak{U}_t^{\mathcal{C}})$ is defined by the least set of the following rules:

$$\begin{aligned} \text{Delay:} \quad & \frac{\check{l} \xrightarrow{\delta, \kappa, \varrho} \hat{l} \quad d \in \mathfrak{U}_t \quad \zeta' = \zeta + d \quad \zeta \models \kappa \wedge \zeta' \models \kappa}{(\check{l}, \vartheta, \zeta) \xrightarrow{d} (\hat{l}, \vartheta, \zeta')} \\ \text{Action:} \quad & \frac{\hat{l} \xrightarrow{g(i), \varphi, \rho} \check{l}' \quad \check{l}' \xrightarrow{\delta, \kappa, \varrho} \hat{l}' \quad \varsigma \in \mathfrak{U}\{i\} \quad \zeta \cup \vartheta \cup \varsigma \models \varphi \wedge \kappa[\rho]}{(\check{l}, \vartheta, \zeta) \xrightarrow{(g, \varsigma(i))} (\check{l}', \vartheta', \zeta')} \end{aligned}$$

with $\vartheta' = ((\vartheta \cup \varsigma)_{ev} \circ \rho)_{\mathcal{V}}$ $\zeta' = ((\zeta)_{ev} \circ \lambda)_{\mathcal{C}}$

4 Towards $\mathbf{stio}_{\mathcal{D}}$

We give DSTS a symbolic trace semantics, which is a prerequisite to define the implementation relation $\mathbf{stio}_{\mathcal{D}}$. In order to do so, we introduce *history interaction/delay variables* which keep track of the possible input- and output-data that a DSTS might send and receive during execution, and of the sojourn times in the locations, respectively. For the set of interaction variables \mathcal{J} , we introduce sets \mathcal{J}_n for all $n \in \mathbb{N}$ such that $i \in \mathcal{J}$ iff $i_n \in \mathcal{J}_n$. Moreover, we introduce sets \mathcal{T}_n for all $n \in \mathbb{N}$ with $d \in \mathcal{T}$ iff $d_n \in \mathcal{T}_n$. We have thus a bijective mapping r_n between all $\mathcal{J}_n \cup \mathcal{T}_n$ and $\mathcal{J} \cup \mathcal{T}$.

Using the history variables, we are able to generalise the switch relation \rightarrow of a DSTS to *traces* of gates. These generalised transitions $l \xrightarrow{\sigma, \varphi, \beta} l'$ describe how a location l' can be reached via a series of delays and interactions over gates. This symbolics semantic for DSTS is the key concept of our framework. We explain the symbolic semantics by means of the example in Figure 2. There will be no formal definition since this would go beyond the scope of this paper.

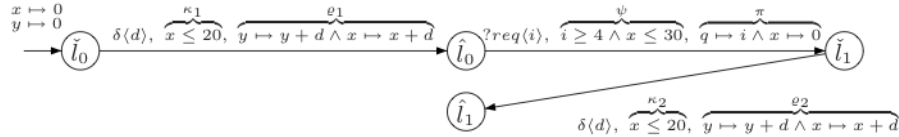


Fig. 2.

In Figure 2 we start with delaying $\delta\langle d \rangle$ in \check{l}_0 . For this the constraint $\kappa_1[\varrho_1] = x + d \leq 20$ has to be satisfied. Note that the clock x has to satisfy

the clock constraint after delaying, as the clock constraint is an invariant in the original *STA*. We then reach \hat{l}_0 , after increasing all clocks with d . Since d is kept symbolic we have to remember the update mapping ϱ_1 we have made and the attainment constraint $\kappa_1[\varrho_1]$ (where $\kappa_1[\varrho_1]$ stands for the substitution of the free variables in κ_1 with ϱ_1) that must be satisfied for reaching \hat{l}_0 . We rename the delay d to history delay variable d_1 by using bijection r_1 . For our generalised transition we then get $\check{l}_0 \xrightarrow{\sigma; \varphi_1, \rho_1} \hat{l}_0$ with $\sigma = \delta$, $\varphi_1 := x + d_1 \leq 20 = \kappa_1([r_1] \circ [\varrho_1])$ and $\rho_1 := y \mapsto y + d_1 \wedge x \mapsto x + d_1 = [r_1] \circ \varrho_1$. Next we take the transitions $?req\langle i \rangle$ from \hat{l}_0 to \check{l}_1 for which the attainment constraint ψ has to be satisfied and π is an update mapping. Additionally, for reaching \check{l}_1 , the constraint $\kappa_2[\pi][\rho_1]$ from the next delay transition (the invariant for $_1$ of the original *STA*) has to be satisfied after updating the clock with $[\pi][\rho_1]$. So we get $(\psi[r_2])[\rho] = i_2 \leq 4 \wedge x + d_1 \leq 20$, $(\kappa_2[\pi])[\rho] = 0 \leq 20$ and $\rho_2 := [\rho_1] \circ [r_2] \circ \pi = q \mapsto i_2 \wedge x \mapsto 0 \wedge y \mapsto y + d_1$ as our new update mapping. For the generalised transition from \check{l}_0 to \check{l}_1 we combine all constraints and update mappings and get $\check{l}_0 \xrightarrow{\sigma_2, \varphi_2, \rho_2} \check{l}_1$, where $\sigma_2 := \sigma_1 \cdot ?req$, $\varphi_2 := \varphi_1 \wedge (\psi[r_2])[\rho] \wedge (\kappa_2[\pi][r_2])[\rho]$, and $\rho_2 := [\rho_1] \circ [r_2] \circ \pi$.

Applying the same rules as before to the switch $\check{l}_1 \xrightarrow{\delta\langle d \rangle, \kappa_2 \varrho_2} \hat{l}_1$, we obtain the generalised transition $\check{l}_0 \xrightarrow{\sigma_2 \cdot \delta\langle d \rangle, \varphi_2 \wedge (\kappa_2[\varrho_2][r_3])[\rho_2], [\rho_2] \circ [r_2] \circ \varrho_2} \hat{l}_1$.

Delayed Symbolic states are tuples (l, φ, ρ) , where φ is the collection of constraints and ρ is the concatenation of update mappings from the generalised transition $\check{l}_0 \xrightarrow{\sigma, \varphi, \rho} l$. For the *DSTS* in Figure 2, $(\hat{l}_0, x \leq 20, y \mapsto y + d_1 \wedge x \mapsto x + d_1)$ is one delayed symbolic state. We write $(\check{l}_0, \top, \text{id})$ for the initial delayed symbolic state with \top being a tautology and id maps all variables to themselves.

In some cases it is necessary to have a constraint on interaction variables and delays occurring in a trace σ . We call such traces *delayed symbolic extended traces*. A delayed symbolic extended trace (σ, χ) consists of a trace $\sigma \in (\mathcal{G} \cup \mathcal{D})^*$ and of a additional constraint χ for history variables of σ .

The set of delayed symbolic states that can be reached after a delayed symbolic extended trace (σ, χ) is defined as follows :

$$(\check{l}_0, \top, \text{id}) \mathbf{after}_{\mathcal{D}}(\sigma, \chi) \triangleq \{(l', \varphi \wedge \chi, \rho) \mid l \xrightarrow{\sigma, \varphi, \rho} l'\}$$

An implementation conforms to a specification if all outputs and delays of the implementation are mimicked by the specification. Therefore, we define the $\mathbf{out}_{\mathcal{D}}$ -set of a symbolic state (l, φ, ρ) , which includes all output actions and delays that can be made from (l, φ, ρ) .

Definition 3 ($\mathbf{out}_{\mathcal{D}}$).

$$\begin{aligned} \mathbf{out}_{\mathcal{D}}((\hat{l}, \varphi, \rho)) &\triangleq \{(g, \varphi, \psi[\rho] \wedge \kappa[\pi][\rho]) \mid \exists \check{l}, \pi : \hat{l} \xrightarrow{g, \psi, \pi} \check{l} \xrightarrow{\delta, \kappa, \varrho} \hat{l}' \wedge g \in \mathcal{G}_U\} \\ \mathbf{out}_{\mathcal{D}}((\check{l}, \varphi, \rho)) &\triangleq \{(\delta, \varphi, \kappa[\varrho][\rho]) \mid \exists \hat{l}, \pi : \check{l} \xrightarrow{\delta, \kappa, \varrho} \hat{l} \wedge \delta \in \mathcal{D}\} \end{aligned}$$

An output action $(g, \varphi, \psi[\rho] \wedge \kappa[\pi][\rho])$ or a delay $(\delta, \varphi, \kappa[\varrho][\rho])$ can be observed if the attainment constraint φ of the delayed symbolic state and the attainment constraint ψ for the actual transition are satisfied.

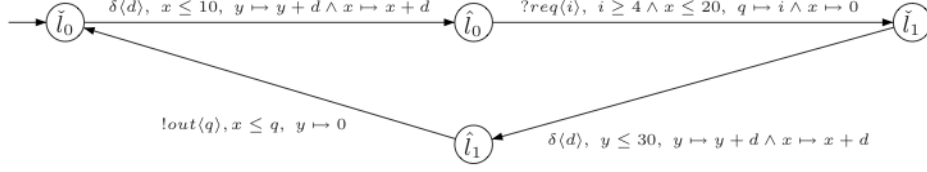


Fig. 3. Example 1

Example 1. Consider $\mathbf{out}_{\mathcal{D}}((\check{l}_0, \top, \mathbf{id})\mathbf{after}_{\mathcal{D}}(\sigma, \chi))$ with $\sigma = \delta?req$ and $\chi = d_1 \leq 6 \wedge i \leq 7$ in Figure 3. Then is (l_1, φ, ρ) the delayed symbolic state reached with $(\check{l}_0, \top, \mathbf{id})\mathbf{after}_{\mathcal{D}}(\sigma, \chi)$ with $\varphi = x + d_1 \leq 10 \wedge i_1 \geq 4 \wedge x + d_1 \leq 20 \wedge i_1 \leq 6$ and $\rho = [y \mapsto y + d \wedge x \mapsto x + d] \circ (q \mapsto i_1 \wedge x \mapsto 0)$. To observe some delay d_2 after \check{l}_1 the constraint $y + d_2 \leq 30$ has to be fulfilled. We already know that $y \mapsto d_1$ so we get $d_1 + d_2 \leq 30$ and $\mathbf{out}_{\mathcal{D}}(\check{l}_1, \varphi, \rho)$ is $(\delta\langle d_2 \rangle, \varphi, d_1 + d_2 \leq 30)$.

Finally, we adapt the definition of **sioco** from Frantzen to *DSTS* in order to define our new implementation relation $\mathbf{stioco}_{\mathcal{D}}$.

Definition 4. Let $\mathcal{S} = (L_{\mathcal{S}}, l_{\mathcal{S}}, \mathcal{V}_{\mathcal{S}}, \mathcal{J}, \mathcal{T}, \mathcal{G}, \mathcal{D}, \mathcal{C}_{\mathcal{S}}, \rightarrow_{\mathcal{S}})$ be an initialised specification, $\mathcal{P} = (L_{\mathcal{P}}, l_{\mathcal{P}}, \mathcal{V}_{\mathcal{P}}, \mathcal{J}, \mathcal{T}, \mathcal{G}, \mathcal{D}, \mathcal{C}_{\mathcal{P}}, \rightarrow_{\mathcal{P}})$ be an input enabled implementation, with $\mathcal{V}_{\mathcal{P}} \cap \mathcal{V}_{\mathcal{S}} = \emptyset$ and \mathcal{F}_s be a set of delayed symbolic extended traces of \mathcal{S} . An implementation \mathcal{P} is $\mathbf{stioco}_{\mathcal{D}}$ conform to a specification \mathcal{S} , written as $\mathcal{P} \mathbf{stioco}_{\mathcal{D}} \mathcal{S}$, if the following holds:

$$\begin{aligned} & \forall (\sigma, \chi) \in \mathcal{F}_s \quad \forall (g \in \mathcal{G}_U \vee g \in \mathcal{D}) : \\ & [\mathcal{C}_{\mathcal{P}} \mapsto 0] \cup [\mathcal{C}_{\mathcal{S}} \mapsto 0] \cup (\iota_{\mathcal{P}})_{\mathcal{V}_{\mathcal{P}}} \cup (\iota_{\mathcal{S}})_{\mathcal{V}_{\mathcal{S}}} \models \bar{\forall}_{\mathcal{J} \cup \hat{\mathcal{J}} \cup \mathcal{T} \cup \hat{\mathcal{T}}} (\phi_{\mathcal{D}}(l_{\mathcal{P}}, g, \sigma) \wedge \chi \rightarrow \phi_{\mathcal{D}}(l_{\mathcal{S}}, g, \sigma)) \\ & \text{where } \phi_{\mathcal{D}}(l, g, \sigma) = \bigvee \{ \varphi \wedge \psi \mid (g, \varphi, \psi) \in \mathbf{out}_{\mathcal{D}}((\check{l}_0, \top, \mathbf{id})\mathbf{after}_{\mathcal{D}}(\sigma, \top)) \} \end{aligned}$$

In the case that $\mathcal{P} \mathbf{stioco}_{\mathcal{D}} \mathcal{S}$, the definition of $\mathbf{stioco}_{\mathcal{D}}$ implies that if the constraint $\varphi \wedge \psi \wedge \chi$ is satisfiable by a certain valuation for a delay or output $(g, \varphi, \psi) \in \mathbf{out}_{\mathcal{D}}((\check{l}_{\mathcal{P}}, \top, \mathbf{id})\mathbf{after}_{\mathcal{D}}(\sigma, \top))$, then there must be a constraint $\varphi' \wedge \psi'$ for a $(g, \varphi', \psi') \in \mathbf{out}_{\mathcal{D}}((\check{l}_{\mathcal{S}}, \top, \mathbf{id})\mathbf{after}_{\mathcal{D}}(\sigma, \top))$ which is also satisfiable by the same valuation. This can be visualised by considering the following abstract example. Let $g \in \mathcal{G}_U$ be output observed from the implementation after a trace σ . We then have $(g, \varphi, \psi) \in \mathbf{out}_{\mathcal{D}}((\check{l}_{\mathcal{P}}, \top, \mathbf{id})\mathbf{after}_{\mathcal{D}}(\sigma, \top))$. This output can be observed if $\varphi \wedge \psi$ is satisfiable by a valuation of variables and delays. We then know that $\phi_{\mathcal{D}}(l_{\mathcal{P}}, g, \sigma)$ is also satisfiable. If the valuation also satisfies the constraint χ , given by the specification, the first part of the implication is true. This then requires that $\phi_{\mathcal{D}}(l_{\mathcal{S}}, g, \sigma)$ is also satisfiable by the same valuation, which implies that there is a constraint $\varphi' \wedge \psi'$ satisfiable by the same valuation. This means that there must be a $(g, \varphi', \psi') \in \mathbf{out}_{\mathcal{D}}((\check{l}_{\mathcal{S}}, \top, \mathbf{id})\mathbf{after}_{\mathcal{D}}(\sigma, \top))$.

We can show that the implementation relation $\mathbf{stioco}_{\mathcal{D}}$ for *DSTS* coincides with **tioco** for *TLTS*.

5 Conclusion

We have presented a testing theory for timed and symbolic testing including the implementation relation $\mathbf{stioco}_{\mathcal{D}}$. This theory is sound and correct with respect to its semantics and can be the basis for future research considering the automatic generation of test cases. Still, our theory does not consider τ -transitions and allows only a certain way of interaction between time and data. This and the investigation on efficient implementation might also be part of future research.

Acknowledgement. I thank Dr. H. Bohnenkamp and Prof. J.-P. Katoen from RWTH Aachen University and Dr. J. Schmaltz from Radboud University.

References

1. J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *LNCS vol 3098*. Springer, 2004.
2. H. C. Bohnenkamp and A. F. E. Belinfante. In *Formal Methods Europe (FME)*, LNCS.
3. L. Brandán Briones. *Theories for Model-Based Testing: Real-time and Coverage*. PhD thesis, University Twente, NL, 2007.
4. L. Brandan Briones and E. Brinksma. A test generation framework for quiescent real-time systems, 2004.
5. L. Frantzen, J. Tretmans, and T.A.C. Willemse. A Symbolic Framework for Model-Based Testing. In *FATES/RV 2006*, 2006.
6. M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *Proc. SPIN 2004*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004.
7. J. Schmaltz and J. Tretmans. On conformance testing for timed systems. In *FORMATS 08*, pages 249–263, 2008.
8. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3):103–120, 1996.

Testing of Hybrid Systems using Qualitative Models

Harald Brandl *

Institute for Software Technology, Graz University of Technology, Austria
brandl@ist.tugraz.at

Abstract. Hybrid systems originate from control theory where the behavior of a discrete controller is observed in its environment. In many cases it is not sufficient to test the control programs alone but also the interactions with their environments. For instance, conclusions over stability or long-term behavior can only be drawn when having a closed-loop view on the system.

The challenge of testing Hybrid Systems is to deal with infinite state spaces of continuous model variables that arise during exploration. By applying predicate abstraction techniques this issue can be countered. This work presents Qualitative Reasoning, which is an Artificial Intelligence technique for common sense reasoning, as a means to abstract continuous behavior to transition systems. Hence, we get a discrete behavior of the hybrid system which we use as model for subsequent test case generation.

1 Introduction

Discrete systems that have continuous interaction with their environment are called hybrid systems. Most embedded systems, e.g. a weather station sensing temperature etc. or a fuel injection controller, are of this kind. When such systems are safety critical, difficult to access, or far away like Mars rovers, it is very important to test them thoroughly. Because of the system complexity of real world applications it is virtually impossible to derive enough test cases by hand in order to meet certain testing coverage metrics. We apply blackbox testing in order to generate test cases due to test purposes or coverage criteria.

Blackbox testing requires a description of the input/output behavior of the implementation under test in some formal notation. We use qualitative models as specification language to describe the continuous behavior of hybrid systems. *Qualitative Reasoning*, an Artificial Intelligence technique for common sense reasoning, can infer behavior from qualitative models by simulation. The models consist of constraints, called *Qualitative Differential Equations (QDEs)*, which are an abstract representation of *Ordinary Differential Equations (ODEs)*. Given a model and an initial state, simulation engines like *QSIM* [1] produces a transition system (TS) which contains all possible behaviors that may evolve over

* Research herein was funded by the EU project ICT-216679, Model-based Generation of Tests for Dependable Embedded Systems (MOGENTES).

time. We use the TS as test model where each state in the TS evaluates all model variables. Hence, the test model has an *Labeled Transition System (LTS)* semantics for which several testing approaches exist.

A behavior is defined as a trace in the LTS, i.e. a sequence of observable events, starting with an initial event. The set of events is called *alphabet*. Due to different abstraction levels the implementation and specification usually do not share the same alphabet. Hence, there must exist a data refinement relation which translates between these alphabets. The translation can be done offline during test case generation. Here, the test cases are directly executable at the implementation level by e.g. linking the test case to the executable. The translation can also be applied online during test case execution via a test adapter. In our work we use a test adapter to execute abstract test cases.

In order to give valid verdicts about test executions, testing assumptions have to be made. One assumption is that the implementation can be represented with the same formalism as the specification. This assumption has to be proved with the data refinement relation. Several proof techniques to show data refinement are presented in [2]. A further assumption is the uniformity hypothesis [3] which states that it is sufficient to choose one input from an equivalence class since the implementation will show the same behavior for all inputs from this class. This assumption is applied in the execution of qualitative test cases [4], where we refine abstract inputs to concrete inputs.

In our first work [5] we describe the mathematical relationship between ODEs and QDEs on a small example and generate test sequences by reachability analysis on the simulation output. In [6,7] we deal with test case generation based on test purposes and the minimization of test cases due to observable events. The work in [8] introduces coverage criteria for qualitative models and describes coverage-based test case generation. We implemented a prototype test case generator, called *QRPathfinder*, which generates controllable test cases according to our newly introduced conformance relation *qriocnf* [4]. In [4] we describe how hybrid systems can be tested with qualitative models and, in a first step, show how test case execution works for a continuous system.

The remainder of this paper is organized as follows: In section 2 we give a brief overview on Qualitative Reasoning. Section 3 deals with the generation of qualitative test cases and their execution. In section 4 we discuss other approaches for testing hybrid systems and relate them to our work. We conclude in section 5 and mention some open issues for future work.

2 Qualitative Reasoning

When modeling using QR, one abstracts the domain of continuous, real valued variables of a system to the domain of points and intervals. A point, called landmark, is a boundary between intervals. In QR the model variables are called *quantities* $q \in Q$ and the domain of a quantity, consisting of landmarks and intervals, is denoted as *quantity space* (QS_q). For instance, when we consider the temperature of water from a qualitative point of view, we can map the

temperature to the domain: below 0°C water is frozen, at 0°C it melts, above 0°C it is liquid, and around 100°C , depending on the pressure it starts boiling. Above 100°C it changes to steam. Hence, the quantity space consists of three intervals and two landmarks, i.e. 0°C and 100°C .

Time is abstracted to a sequence of temporally ordered states of behavior where each state binds all quantities to values. The value of a quantity is a pair consisting of an abstract value $qVal \in QS_q$ from its quantity space and the direction of change δ . The direction of change $\delta =_{af} \{-, 0, +\}$ is an abstraction of the first derivation $\frac{\partial}{\partial t}$ where δ is either decreasing, steady, or increasing. The behavior of a QR model changes from one state to the next if one of the quantities alters its value, i.e. there is a change of $qVal$ or δ .

Qualitative models, consisting of a set of QDEs, are constraint satisfaction problems (CSPs). In a worst case scenario, i.e. an unconstrained system, the size of the transition system is simply the whole state space: $\times_{q \in Q} |\delta| \cdot |QS_q|$.

3 Testing with QR Models

Test Case Generation We described in [6,7] the test case generation methodology based on qualitative models. In a first step the test engineer builds the model and validates it by evaluating the simulation result. When testing with test purposes the engineer formulates a scenario of interest in some notation, in our case as regular expression over symbolic labels. Each label expresses a property over the set of quantities. Then the simulation result, denoted as QR transition system (QR TS), is labeled with the property symbols. Each transition in the resulting LTS contains the labels which are satisfied in the following state. The labeling results in a Kripke structure from which we compute the synchronous product with the deterministic finite automaton (DFA) that is equivalent to the regular expression stating the test purpose. The product TS is a slice of the specification containing the behavior of interest.

Next, we minimize the product TS due to disjoint input quantities L_I and output quantities L_U that are relevant for testing. The remaining quantities are hidden and their evolution cause internal (τ) transitions. We refer to the set of observable quantities as $L =_{af} L_I \cup L_U$. During weak-bisimulation minimization the resulting TS may get non-deterministic which, in a subsequent step, is converted via standard subset construction into a deterministic TS, called Complete Test Graph (CTG). Note that the determinization has no influence on the observable behavior and hence on the conformance relation. From this CTG we extract controllable test cases according to our *qrioconf* relation, see Definition 1.

Definition 1.

$$i \text{ qrioconf } s =_{af} \forall \sigma \in \text{Traces}(s) \downarrow L_I : \\ \text{out}^{\sim}(i \text{ after }^{\sim} \sigma) \subseteq \text{out}^{\sim}(s \text{ after }^{\sim} \sigma)$$

Here i denotes the implementation, s is the specification, $Traces(s)$ is the set of sequences of states in the specification starting from the initial state, σ is an input trace (i.e. a trace projected on L_I), \mathbf{after}^{\sim} determines the set of states after σ in i and s respectively, and \mathbf{out}^{\sim} evaluates the output quantities of a set of states.

The conformance relation states that for all input traces in the specification the following must hold: The outputs of the implementation after the input trace have to be a subset of the outputs of the specification after the same trace. Figure 1 shows three QR transition systems where we use integers as qualitative

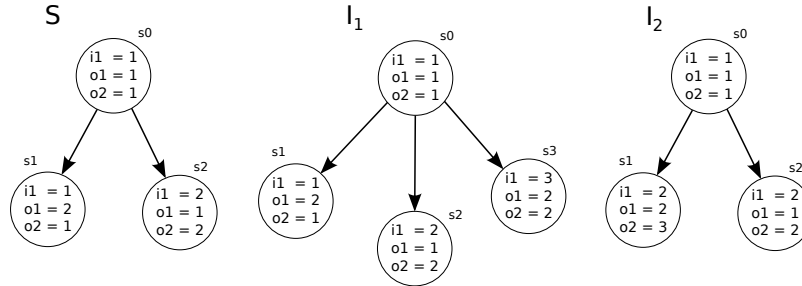
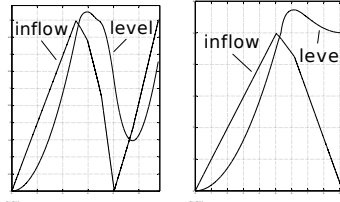


Fig. 1. Conformance between QR transition systems: I_1 $qrioconf$ S and I_2 $qrioconf$ S

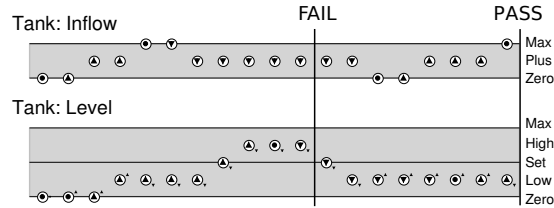
values. In the following we refer to the values of inputs and outputs within a state in a vectorized form, $[i_1, \dots, i_n]^T$ and $[o_1, \dots, o_m]^T$ respectively. The states s_0 of I_1 and I_2 are $qrioconf$ to S since $I_1 \mathbf{after}^{\sim} \langle [1] \rangle = I_2 \mathbf{after}^{\sim} \langle [1] \rangle = \{s_0\}$ and the outputs $\mathbf{out}^{\sim}(s_0) = \{[1, 1]^T\}$ are allowed in S after the same input trace $\langle [1] \rangle$. The outputs in I_1 after the input traces $\langle [1], [1] \rangle$ and $\langle [1], [2] \rangle$ are both allowed in the specification S . For the third input trace $\langle [1], [3] \rangle$ the last input $i_1 = 3$ is not specified in S . Due to implementation freedom the behavior following unspecified inputs is not considered by the conformance relation. For I_2 we get $I_2 \mathbf{after}^{\sim} \langle [1], [2] \rangle = \{s_1, s_2\}$. However, the outputs $\mathbf{out}^{\sim}(\{s_1, s_2\})$ of I_2 are not a subset of $\mathbf{out}^{\sim}(S \mathbf{after}^{\sim} \langle [1], [2] \rangle)$. Hence I_2 is not $qrioconf$ S .

Test Case Execution In [4] we describe the execution of qualitative test cases and demonstrate it on a small water tank example. The tank is filled from the top and it has one outlet on a certain height and another one at the bottom. The system has four different modes, i.e. the water level is above or below the upper outlet and the lower outlet may be open or closed. The aim is to test, if a concrete instance of the water tank conforms to the specification model. We used a *Matlab Simulink* model of the tank as implementation under test. Since the specification is on a more abstract level than the implementation we have to deal with data refinement during testing. Therefore we apply *U-Simulation*, described in [2], where we refine abstract inputs from the specification to the

implementation level and then abstract the observed outputs from the implementation back to the specification level. Hence, during testing we map between continuous functions and qualitative traces. Figure 2 depicts the execution of a test case where there is some inflow into the tank and the lower outlet is open. In the test case *inflow* is an input and *level* is an output. One can observe, that the



(a) The Implementation passes the Test Case.
 (b) The Mutant fails the Test Case.



(c) Trace of the Test Case executed on the Implementation (PASS) and the Mutant (FAIL).

Fig. 2. Execution of a Test Case.

qualitative input trace is refined to a piecewise linear function. The definition of the slope of the input is part of the specification as well as other timing information like the sampling interval or timeouts. While applying input samples to the implementation the observed output samples undergo qualitative abstraction. The resulting qualitative values must be allowed by the specification which is the case for the correct tank instance. In the wrong tank instance, called *Mutant*, two modes of the system are exchanged, i.e. when the valve at the bottom is in opened position it is closed actually. Figure 2 shows that the test case discovers the mutant by giving a *fail* verdict.

4 Related Research

We apply similar techniques for test case generation with test purposes as the tool *TGV* [9], a test case generator from labeled transition systems.

The work in [10] presents symbolic test case generation for *time-discrete input-output hybrid systems (TDIOHS)* specifications. In contrast to our work such specifications already contain the numerical solutions of the continuous behaviors of the system. They developed a constraint solver for handling non-linear CSPs with real-valued variables based on interval analysis techniques.

The authors in [11] propose a test case generation algorithm for continuous and hybrid systems that relies on *Rapidly-exploring Random Trees (RRTs)*. They can handle high-dimensional search spaces and generate test cases due to certain coverage criteria.

The main difference to existing abstraction techniques for hybrid systems is that we solve abstract equations (QDEs) rather than concrete, real-valued equations (ODEs). In the latter case polyhedral constraints are used to enclose regions of possibly infinite states within which the real solution is located. QR provides a more abstract view on a system where constraints are not defined over real values but over the finite domain of landmarks and intervals.

5 Conclusion and Future Work

In our previous work we elaborated the possibility the apply qualitative models for testing continuous systems. We showed how to generate controllable test cases with test purposes [6,7] and according to coverage criteria [8]. Because of qualitative abstraction the degree of nondeterminism can lead to big state spaces. Online testing, where the state space is explored on-the-fly, will be an interesting subject for future work.

Furthermore we developed abstraction and refinement techniques in order to execute qualitative test cases. The approach has been demonstrated with a prototype tool. As a next step we will adapt some hybrid formalism where we replace the description of continuous behavior with qualitative behavior.

References

1. Kuipers, B.: Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge. MIT Press (1994)
2. DeRoeper, W., Engelhardt, K.: Data Refinement: Model-Oriented Proof Methods and Their Comparison. Cambridge University Press, New York, NY, USA (1999)
3. Gaudel, M.C.: Testing can be formal too. In: TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE. Volume 915 of Lecture Notes in Computer Science., Springer-Verlag (1995) 82–96
4. Aichernig, B.K., Brandl, H., Wotawa, F.: Conformance testing of hybrid systems with qualitative reasoning models. In Finkbeiner, B., Gurevich, Y., Petrenko, A.K., eds.: Model-Based Testing, MBT 2009. (2009) 45–59
5. Brandl, H., Wotawa, F.: Test case generation from qr models. In: 21st International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems. (2008)
6. Brandl, H., Fraser, G., Wotawa, F.: QR-model based testing. In: AST'08: Proceedings of the 3rd international workshop on automation of software test, NY, USA, ACM (2008) 20–17

7. Brandl, H., Fraser, G., Wotawa, F.: A report on QR-based testing. In: 22nd International Workshop on Qualitative Reasoning. (2008) 1–9
8. Brandl, H., Fraser, G., Wotawa, F.: Coverage-based testing using qualitative reasoning models. In: Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), Knowledge Systems Institute Graduate School (2008) 393–398
9. Jard, C., Jeron, T.: TGV: theory, principles and algorithms. International Journal on Software Tools for Technology Transfer (STTT) **7** (2004) 297–315
10. Badban, B., Fränzle, M., Peleska, J., Teige, T.: Test automation for hybrid systems. In: SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance, New York, NY, USA, ACM (2006) 14–21
11. Dang, T., Nahhal, T.: Coverage-guided test generation for continuous and hybrid systems. Form. Methods Syst. Des. **34** (2009) 183–213

Formal Domain Modeling: From Specification to Validation ^{*}

Atif Mashkoor

LORIA – DEDALE Team – Nancy Université
Campus Scientifique, BP 239,
F-54506, Vandœuvre lès Nancy, France
`{firstname.lastname}@loria.fr`

Abstract. The main theme of this research is to study and develop techniques for the modeling of software controlled safety critical systems. In this work, we formally specify different entities, phenomena and their inter-relationships, and specially non functional properties related to land transportation systems with refinement based approach at domain level. We also introduce a stepwise validation process to maintain seamlessness between environment and its captured models. We apply our results on two safety critical case studies.

1 Introduction

Having good understanding of an application domain is a crucial prerequisite to develop software within that domain. The understanding of a domain is referred to as a domain model. A domain model is a conceptual model of a system which describes various entities, phenomena and their inter-relationships, along with their important static and dynamic properties of the domain. The domain model may be expressed in the form of requirements, specifications, or architectural references.

According to [1], if domain models and requirements of software are not formally expressed, software correctness can not be meaningfully achieved. Safety is also one of the major factors which can not be overlooked while designing complex and critical systems. The development of correct and safe systems can be difficult and error prone with traditional software development methods. However, use of formal methods, in order to ensure their correctness and to structure their development from domain modeling to implementation, can significantly help system development.

Formal languages are notoriously difficult to read for the non-initiated. Furthermore, well-written specifications often introduce abstract objects and operations that have no intuitive concrete counterpart. Hence, validation has to wait.

^{*} This work has been partially supported by the ANR (National Research Agency) in the context of the TACOS project, whose reference number is ANR-06-SETI-017 (<http://tacos.loria.fr>), and by the Pôle de Compétitivité Alsace/Franche-Comté in the context of the CRISTAL project (<http://www.projet-cristal.org>).

This implies that the development of the model requires an uncomfortable level of trust.

The pivotal concept of formal methods such as B [2] is the notion of refinement and its relation to correctness. The assessment of the correctness of a piece of code, its verification, is no more a unique big process step but it is broken down into small pieces along with the whole development process. The proof of correctness is the sum of the proofs of small assertions (invariant preservation, well-formedness, existence of abstraction function, etc.) associated to each refinement. Likewise, the technique of animation could be used with each refinement step to break its validation into smaller assessments.

Introduction of validation into refinement based processes yields two advantages: First, early detection of problems in the models (say, misunderstanding about a certain behavior) should be easier and inexpensive to correct. Second, users can be involved into the development right from the start.

In this work, we focus on animation technique which is the “execution” of a specification as a mean to validate it. Thanks to the development of tools, like Brama [3], it is possible to animate specifications in B or Event-B [4] before they reach an implementation stage. However, there are restrictions on the kind of specifications that can be animated. Non-constructive definitions, infinite sets, or complex quantified logic expressions are among the list of restrictions.

We devise a technique to animate abstract specifications by systematic transformations. The product of the transformations is a specification which may be non provable, but which is guaranteed to have the same behavior as the formally correct initial specification. This goal is achieved through the design of a set of transformation heuristics whose correction is rigorously asserted and a rigorous process.

The main aim of the research is two fold: first, formal modeling of data, behaviors, protocols, interaction between elements, and non-functional properties of transportation domain and second, to incorporate a stepwise validation technique into the overall modeling process.

The presentation of the paper is organized as follows: Next section presents the language and tool we use: Event-B and Brama. Then we present a stepwise specification and validation process for domain modeling. Two case-studies are described thereafter to show how we have implemented our approaches. Finally, we conclude that what should now be completed to have a technique that could be used as standard practice and our future research endeavors.

2 Tools

This section introduces the tools which we use for specification and validation.

2.1 Event-B

Event-B is a formal method for system-level modeling and analysis of large reactive and distributed systems. Main features of Event-B are the use of set theory

and first order logic for modeling systems, the use of refinement to represent systems at different levels of abstraction and the use of mathematical proof to verify consistency between refinement levels. Event-B is provided with tool support in the form of an Eclipse-based IDE called RODIN¹ which is a platform for writing and proving Event-B specifications.

2.2 Brama

Brama is an animator for Event-B specifications. It is an Eclipse based plugin for the RODIN platform which can be used in two complementary modes. Either Brama can be manually controlled from within the RODIN or it can also be connected to a Flash graphical animation through a communication server; it then acts as the engine which controls the graphical effects.

3 Stepwise specification and validation

In order to verify complex systems against their requirements, the breakdown of overall system into different levels of abstraction is highly recommended. This can be achieved by their stepwise development. To specify our domain models we follow the same principle. We start from abstract requirements, we gradually refine them to achieve concrete and fine grained description of the model, and verify each refinement step against the specification constructed in the previous refinement.

We use two different notions of refinement to specify our models: horizontal refinement and vertical refinement. In horizontal refinement the details are added to the model while remaining at the same level of abstraction. However in vertical refinements we add the description to the model while making a leap to the next abstraction level.

Once a domain model has been specified and verified, an important question arises: does it accurately capture the environment? While proof tools guarantee the consistency of the specification, they are of little help to check if it is the true representation of the environment. Like the verification of the model can be broken down into smaller proofs associated with each refinement step, we use the technique of animation at each refinement step to break its validation into smaller assessments.

Animation of specification is not that straightforward because it heavily depends upon tools. Any limitation of the tool will be a restriction on the class of animatable specifications. To validate a specification which does not belong to this class, we need to “bring it in.” We do this by applying transformation rules which are designed to keep the behavior unaltered, possibly at the expense of other properties.

¹ <http://sourceforge.net/projects/rodin-b-sharp/>

Experimenting with Brama on two Event-B safety critical systems—a formal domain model of land transportation [5] and a situated multi-agent platooning system [6]—we have dressed up a typology of five general cases:

- 1 Brama forbids finite clauses in axioms
- 2 Brama interprets quantifications as iterations
- 3 Brama cannot compute dynamic functional bindings in substitutions
- 4 Brama does not compute analytically defined functions
- 5 Brama has limited communication with its external graphical environment

This led us to design 10 transformation heuristics [7] expressed following the pattern shown by figure 1.

We designed the heuristics to preserve the behavior of the specification, *not* its formal properties. In particular, the transformed specification may not be provable. The correctness of the transformation is then a crucial issue.

Since heuristics cannot be “proven” within B formal logic system, we relied on the mathematical tradition of *rigorous arguments*. For this to work, we need a basic assumption: the initial specification text must have been formally verified. Most of the arguments given in the **Justification** clause of heuristic rely on this hypothesis.

A *verified* specification must be the starting point of the validation process. The application of the heuristics will “downgrade” it to a non provable specification. Running the animation may uncover some mistakes. These entail the modification of the *initial* specification, which then must be verified, and transformed again for proceeding with the validation. This is summed-up in figure 2.

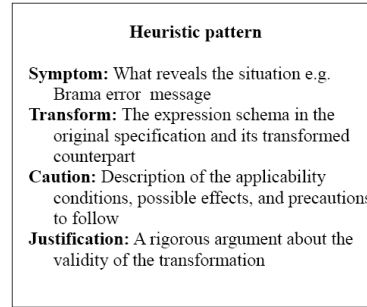


Fig. 1. The heuristic pattern

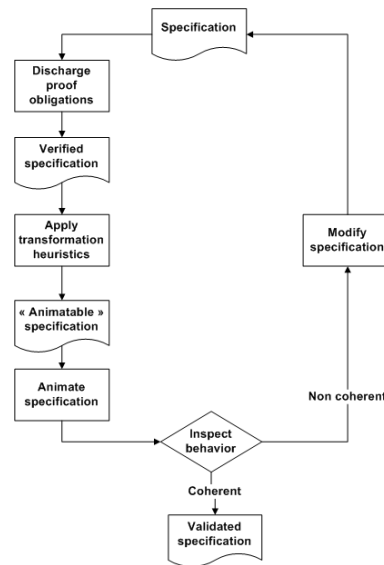


Fig. 2. The validation process

4 Case studies

This section describes two case studies which were the incentive for this work. Both specifications concern the domain of land transport systems. They are

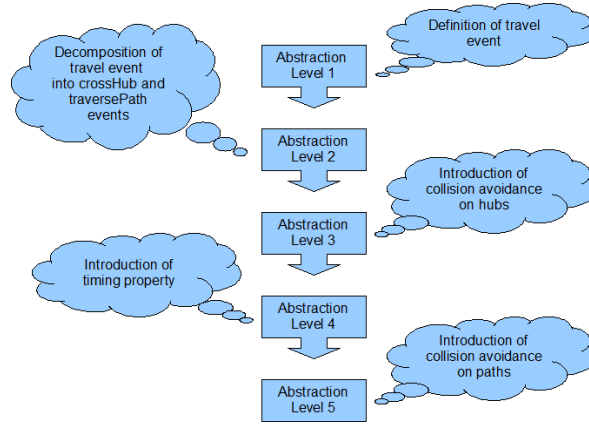


Fig. 3. Levels of abstraction of transport domain model

part of cooperative projects. TACOS is an effort to integrate components and non functional properties into formal requirement specifications. In particular, safety critical properties must be assessed and formalized. CRISTAL is a joint project with the industrial goal of designing urban mobility systems based on autonomous vehicles. As these systems interact with humans and operate on public space, the certification issue is a major problem.

4.1 TACOS

The specification in this case study is about the modeling of the land-transportation domain. In the model, we want to express properties that any system working within the domain is expected to meet and maintain.

In this specification effort, the focus is on the formal definition of concepts, constraints and properties, rather than on the implementation of a particular system. Refinement is then used to introduce new notions; the proof obligations serve to guarantee the consistency of the model. Our devised stepwise validation technique was used for the validation of the model whose details can be found in [8].

The current specification consists of 8 refinements (3 horizontal and 5 vertical). It is organized into five abstraction levels which are summarized by figure 3. A detailed description of the model can be found in [5].

This specification exhibits several properties which call for validation, namely:

- complex data which constraint behaviors (following a route),
- protocols and iterations (travel as sequence of stages, hub crossing protocol), and
- non deterministic interaction between elements (autonomous vehicles).
- several non-functional properties, such as collision avoidance, timing, etc.

4.2 CRISTAL

The second case study deals with a specification of *platooning*. Platooning is a mode of moving where vehicles are synchronized and follow one another closely. A platoon can be seen as a road-train where cars are linked by software instead of hardware. A local Event-B specification of the model has been written [6] as an effort to make it amenable to the formal techniques required by certification.

The specification consists of five machines (one abstract and four vertical refinements). Contrary to the previous case study, the structure of the development of this case study can be interpreted as a sequence of refinements toward an implementation. Each refinement decomposes some events to make explicit a part of the general computation.

We mainly use this case study to experiment our devised transformation heuristics for its validation. The underlying rationale was to observe the animation of a model based on kinematic laws specified by heavy use of functions and also to compare the results of the validation by animation with the results of validation by simulations that had been previously made.

5 Conclusion and future work

Using a modeling language which is not conceived particularly for domain engineering was a challenging task. Though we stumbled upon some shortcomings in Event-B (for example, lack of temporal constraints, lack of notion of sequences, etc.) yet, the general philosophy has been well suited to our purpose. The notions of events and non determinism allow us easy modeling of independent vehicles without any assumption other than their common property: they move. The strong safety constraints we have considered are also easily modeled. Modeling of other non functional properties, such as, collision avoidance, timing, etc. also did not pose great difficulties. All was done through standard refinement techniques. We are thus encouraged to proceed further with enrichment of current domain models specially with inclusion of more non functional properties, such as, oscillation, hooking/unhooking, etc.

While arguing about the relationship between refinement based modeling and its stepwise validation, we discovered that not every refinement step is animatable. This is consistent with using animation as a kind of quality-assurance activity during development. We believe that one animation per abstraction level is sufficient. In fact, the first refinement of a level may often have a non-determinism too wide to allow for meaningful animation (concept introduction), but subsequent refinements get the definitions of the new concept precise enough to allow animation.

The list of heuristics is not closed yet. In future this is expected to grow as we model and validate new properties of the domain. Manual application of these heuristics to specification is tedious, cumbersome and may be error prone if not applied carefully. Therefore we are planning to write a plug-in/tool which can apply these transformations automatically to specifications.

References

1. Bjørner, D.: Development of Transportation Systems. In: International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA) (2007)
2. Abrial, J.R.: The B Book. Cambridge University Press (1996)
3. Servat, T.: BRAMA: A New Graphic Animation Tool for B Models. In: B 2007: Formal Specification and Development in B, Springer-Verlag (2006) 274–276
4. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2009)
5. Mashkoo, A., Jacquot, J.P., Souquières, J.: B Événementiel pour la Modélisation du Domaine: Application au Transport. In: Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'09), Toulouse, France (2009) 1–19
6. Lanoix, A.: Event-B Specification of a Situated Multi-Agent System: Study of a Platoon of Vehicles. In: 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE Computer Society (2008) 297–304
7. Mashkoo, A., Jacquot, J.P.: Incorporating Animation in Stepwise Development of Formal Specification. Research Report INRIA-00392996, LORIA, Nancy, France (2009) <http://hal.inria.fr/inria-00392996/en/>.
8. Mashkoo, A., Jacquot, J.P., Souquières, J.: Transformation Heuristics for Formal Requirements Validation by Animation. In: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems - SafeCert'09, York, UK (2009)

Expressing KAOS Goal Models with Event-B ^{*}

Abderrahman Matoussi

Laboratory of Algorithmic, Complexity and Logic (LACL)
Université Paris-Est
61, avenue Général de Gaulle
94010 Créteil Cedex, France
`abderrahman.matoussi@univ-paris12.fr`

Abstract. In the system software development life-cycle, the first phase corresponds to the requirements engineering. It is followed by the specification phase and then, the development stage. The Event-B method has shown that it was very relevant for the last two phases. So, we aim at introducing requirements analysis into the Event-B method. Thus, it will be possible to prove the requirements model and to establish formal links between this model and the specification of a system.

Keywords. Requirements engineering, KAOS method, Event-B method.

1 Introduction

With most of formal methods, an initial mathematical model can be refined in multiple steps, until the final refinement contains enough details for an implementation. Most of the time, the initial model is derived from the description, obtained by the requirements analysis. Consequently, the major remaining weakness in the development chain is the gap between textual or semi-formal requirements and the initial formal specification. There is little research on reconciling the requirements phase with the formal specification phase. In fact, the validation of this initial formal specification is very difficult due to the inability to understand formal models (for customers) and to link them with initial requirements (for designers). Our objective is to combine the requirements and the specification phases by using KAOS and the Event-B method. On one hand, KAOS is a goal-oriented methodology for requirements engineering which allows analysts to build requirements models and to derive requirements documents. On the other hand, Event-B is a model-based formal method which provides language, techniques and tools to support the analysis and design of systems, from the specification to the implementation stages. Existing work [5,6,8] that combine KAOS with formal methods generate a formal specification model from a KAOS requirements model. Contrary to these methods that take only a subset of the KAOS models into account, our long-term objective is to express the whole KAOS requirements model with Event-B, in order to formally reasoning

^{*} The work in this paper is partially supported by the TACOS project ANR-06-SETI-017 founded by the french ANR (National Research Agency).

about it. The key idea is to stay at the same abstraction level as KAOS. In this paper, we begin by considering the first stage of KAOS requirements analysis, namely the goals modeling. This paper continues our previous works [7] by addressing the Event-B formalization of KAOS patterns with additional studies, results and proofs. The Event-B formalization of the other KAOS models is a work in progress.

The remainder of the paper is organized as follows. Section 2 overviews the KAOS and the Event-B methods that are employed in the proposed approach. Section 3 details our proposed approach that consists in expressing a KAOS goal model with Event-B. Section 4 illustrates the approach by presenting the Event-B formalization of the Exclusive-OR goal refinement pattern. Finally, Section 5 discusses related work and concludes with an outline of future work.

2 Background

In this section, we briefly introduce KAOS and Event-B.

2.1 KAOS method

KAOS (Knowledge Acquisition in autOMated Specification) [4] is a goal-based requirements engineering method. KAOS requires the building of a data model in UML-like notation. A goal defines an objective the system should meet, usually through the cooperation of multiple agents such as devices or humans. KAOS is composed of several sub-models related through inter-model consistency rules: (i) the central model is the *goal model* which describes the goals of the system and its environment; (ii) the *object model* defines the objects (agents ,entity...) of interest in the application domain; (iii) the *agent responsibility model* takes care of assigning goals to agents in a realisable way; (iv) the *operation model* details the operation an agent has to perform to reach the goals he is responsible for.

Goals are organized in a hierarchy obtained from the AND/OR refinement of higher level goals into lower-level goals. Higher-level goals are strategic and coarse-grained while lower-level goals are technical and fine-grained (more operational in nature). KAOS provides a catalog of “Goal Patterns” that generalize the most common goal configurations: (i) *Achieve Goals* desire goals achievement some time in the future; (ii) *Maintain Goals* express that goals must hold at all times in the future; (iii) *Cease Goals* disallow goals achievement some time in the future; (iv) *Avoid Goals* ensure that goals must not hold at all times in the future. KAOS also provides a criterion for stopping the refinement process. If a goal can be assigned to the sole responsibility of an individual agent, there is no need for further goal refinement to occur. Operational goals (goals that are assigned to agents) are the leaves of a goal graph. Each leaf can be either a requirement (if it is assigned to an agent of the system) or an expectation (if it is assigned to an agent in the environment).

2.2 Event-B method

Event-B [2,3] is a formal method for modeling discrete systems by refinement. It is the successor of the B Method [1]. An Event-B model can be described in terms of two basic constructs: (i) *the context* which contains the static part of a model such as carrier sets, constants, axioms and theorems; (ii) *the machine* which contains the dynamic part such as variables, invariants, theorems, events and variants. During the refinement, events are refined to take new variables into account. This is performed by strengthening their guards and adding substitutions on the new variables. New events that only assign the new variables may also be introduced. Proof obligations (POs) are generated to ensure the consistency of the abstract model and the correctness of the refinement. Event-B is supported by several tools, currently in the form a platform called Rodin¹.

3 The proposed approach

3.1 Motivation

Contrary to other requirements methods such as i* [9], KAOS is promising in that it can be extended with an extra step of formality which can fill in the gap between requirements and the later phases of development. Hence, we aim to explore this advantage by expressing KAOS requirements model with the Event-B language. The choice of Event-B is due to its similarity and complementarity with KAOS. Firstly, Event-B is based on set mathematics with the ability to use standard first-order predicate logic facilitating the integration with the KAOS requirements model that is based on first-order temporal logic. Secondly, both Event-B and KAOS have the notion of refinement (constructive approach). Finally, KAOS and Event-B (conversely to the classical B) have the ability to model both the system and its environment.

Since goals play an important role in requirements engineering process and provide a bridge linking stakeholder requests to system specification, the proposed approach comes down to automatically derive Event-B representation from KAOS goal model rather than from KAOS requirements model as a whole. However, it is not possible to verify that both models are equivalent. The Event-B expression of the KAOS goal model allows us to give it a precise semantics.

3.2 Formalization of KAOS Achieve Goals

To achieve our objective, we formalize with Event-B the KAOS refinement patterns that analysts use to generate a KAOS goal hierarchy. In this paper, we focus on the most frequently used "Goal Patterns" : the *Achieve goals*. Formalization of the other categories of goal patterns is a work in progress. The assertions in *Achieve goals* are of the following form: $G\text{-Guard} \Rightarrow \diamond G\text{-PostCond}$,

¹ <http://rodin-b-sharp.sourceforge.net>

where $G\text{-Guard}$ and $G\text{-PostCond}$ are predicates. Symbol \Rightarrow denotes the classical logical implication. Symbol \diamond (the open diamond) represents the temporal operator "eventually" which ensures that a predicate must occur "at some time in the future". Hence, such assertions state that from a state in which $G\text{-Guard}$ holds, we can reach sooner or later another state in which $G\text{-PostCond}$ holds.

If we refer to the concepts of guard and postcondition that exist in Event-B, a KAOS goal can be considered as a postcondition of the system, since it means that a property must be established. The crux of our formalization is to express each KAOS goal as a B event, where the action represents the achievement of the goal. Then, we will use the Event-B refinement relation and additional custom-built proof obligations to derive all the subgoals of the system by means of B events. One may wonder whether the formalization of KAOS target predicates (i.e. the predicate after the diamond symbol) as B postconditions is adequate, since the execution of B events is not mandatory. At this very high level of abstraction, there is only one event for representing the parent goal. In accordance with the Event-B semantics, if the guard of the event is true, then the event necessarily occurs. For the new events built by refinement and associated to the subgoals, we guarantee by construction that no events prevent the postconditions to be established. For that, we have proposed an Event-B semantics for each KAOS refinement pattern by constructing set-theoretic mathematical models. Based on the classical set of inference rules from Event-B [2], we have identified the systematic proof obligations for each KAOS goal refinement pattern. To better illustrate the approach, the next section presents the Event-B refinement semantics related to the *Exclusive-OR goal refinement pattern*.

4 The Exclusive-OR goal refinement pattern

4.1 Description of the KAOS pattern

An Achieve goal G of the form $(G\text{-Guard}) \Rightarrow \diamond(G\text{-PostCond})$ is Exclusive-OR refined into two sub-goals G_1 and G_2 if only one (not both) of its sub-goals is achieved. A typical Exclusive-OR is shown in Figure 1 (with just two sub-goals).

4.2 Formal semantics of the pattern

4.2.1 Formal definition As explained in the last section, each level i ($i \in [0..n]$) is represented in the hierarchy of the KAOS goal graph as an Event-B model M_i that refines the model M_{i-1} related to the level $i - 1$. Moreover, we represent each goal as a B event where the guard is the transcription of $G\text{-Guard}$ from the KAOS goal expression, and the **then** part is the translation into Event-B of $G\text{-PostCond}$ (see Figure 2).

Based on the definition of the Exclusive-OR refinement, we propose to refine the abstract event **EvG** as follows:

$$(\mathbf{EvG1} \text{ XOR } \mathbf{EvG2}) \text{ Refines } \mathbf{EvG}$$

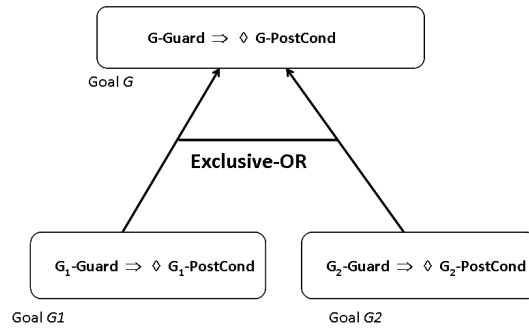


Fig. 1. Exclusive-OR goal refinement pattern

$\mathbf{EvG} \triangleq$ $\mathbf{when} \ G\text{-Guard}$ $\mathbf{then} \ G\text{-PostCond}$ \mathbf{end}	$\mathbf{EvG1} \triangleq$ $\mathbf{when} \ G_1\text{-Guard}$ $\mathbf{then} \ G_1\text{-PostCond}$ \mathbf{end} $\mathbf{EvG2} \triangleq$ $\mathbf{when} \ G_2\text{-Guard}$ $\mathbf{then} \ G_2\text{-PostCond}$ \mathbf{end}
(a) Abstract Model M_0	(b) Refinement Model M_1

Fig. 2. Overview of the Event-B representation of the KAOS goal model

4.2.2 Proof obligations identification Based on the notions of "trace comparisons" and "forward simulation" [3,10], we are going to give systematic rules defining exactly what we have to prove for this pattern in order to ensure that each concrete event (**EvG1**, **EvG2**) indeed refines its abstraction **EvG**. In fact, we have to prove seven different lemmas:

- *The feasibility refinement* must be verified for each concrete event.

$$I(v) , J(v, w) , G_1\text{-Guard}(w) \vdash \exists w' . G_1\text{-PostCond}(w, w') \quad (\mathbf{PO1})$$

$$I(v) , J(v, w) , G_2\text{-Guard}(w) \vdash \exists w' . G_2\text{-PostCond}(w, w') \quad (\mathbf{PO2})$$

- *The guard strengthening* ensures that each concrete guard is stronger than the abstract one. In other words, it is not possible to have the concrete version enabled whereas the abstract one would not. The term "stronger" means that the concrete guard implies the abstract guard.

$$G_1\text{-Guard} \Rightarrow G\text{-Guard} \quad (\mathbf{PO3})$$

$$G_2\text{-Guard} \Rightarrow G\text{-Guard} \quad (\mathbf{PO4})$$

- *The correct refinement* ensures that each concrete event transforms the concrete variables in a way which does not contradict the abstract event.

$$G_1\text{-PostCond} \Rightarrow G\text{-PostCond} \quad (\mathbf{PO5})$$

$$G_2\text{-PostCond} \Rightarrow G\text{-PostCond} \quad (\mathbf{PO6})$$

- We can execute just one event (either **EvG1** or **EvG2**) but not both. This is done by an exclusive disjunction between the different concrete guards.

$$G_1\text{-Guard} \otimes G_2\text{-Guard} \quad (\mathbf{PO7})$$

4.3 Synthesis

The Event-B refinement semantics of the Exclusive-OR goal refinement pattern requires to prove six proof obligations ((**PO1**)...(**PO6**)) that could easily be discharged by the current version of the Rodin automatic theorem prover. In fact, this Event-B refinement semantics is exactly the same one proposed by Rodin if we consider that each event refines the abstract event **EvG**. To express the "exclusive" characteristic of the KAOS refinement, we add a seventh proof obligation ((**PO7**)) in the form of an Event-B theorem.

5 Conclusion and related work

Our proposed approach aims at establishing a bridge between the non-formal (KAOS) and the formal worlds as narrow and concise as possible. In the sequel, we outline some approaches that have tried to bridge the gap between KAOS requirements model and formal methods.

A practical solution for traceability between KAOS requirements and B has been proposed by [8]. It presents a goal-oriented approach to elaborate a pertinent model and turn it into a high quality abstract B machines. The authors of [6] provide means for transforming the security requirements model built with KAOS to an only one abstract B model which can be later refined. The GOPCSD (Goal-oriented Process Control System Design) tool [5] is an adaptation of the KAOS method that serves to analyze the KAOS requirements and generate B formal specifications.

Nevertheless, the reconciliation presented by all of these works remains partial because they don't consider all the parts of the KAOS goal model but only the requirements (operational goals). Consequently, the formal model do not include any information about the non-operational goals and the type of goal refinement. In this paper, we have explored how to cope with this problem using a constructive approach (driven by goals) showing that it is possible to express KAOS goal models with a formal method like Event-B by staying at the same abstraction level. We show also that extending KAOS with more formality in a development framework like Event-B allows requirements to be traced at the various steps of development. Moreover, the main contribution of our approach is that it balances the tradeoff between complexity of rigid formality (Event-B method) and expressiveness of semi-formal approaches (KAOS). So, what we present can be very useful in practice to (i) systematically verify that all KAOS requirements are represented in the Event-B model; (ii) systematically verify that each element in the Event-B model has a purpose in KAOS.

However, a number of future research steps are ongoing. Further work will consist in applying the approach on a number of case studies in order to support non-functional goals. This would address issues of conflict between these goals, which does not exist between functional goals. Moreover, we would like to establish the correspondence between the obtained Event-B representation of KAOS goal models and the later phases of development. At tool level, we plan to develop a connector between KAOS toolset and the RODIN open platform.

Acknowledgment

I would like to thank my Ph.D supervisors Régine Laleau and Frédéric Gervais for their constructive comments and fruitful discussions that helped a lot for the improvement of this work.

References

1. J.R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge U.P, 1996.
2. J.R. Abrial. *Chapter 2 of the forthcoming book: "Modeling in Event-B: System and Software Engineering Forthcoming book"*. http://www.event-b.org/A_ch2.pdf.
3. C. Métayer and J.R. Abrial and L. Voisin. *Event-B Language, RODIN Deliverable D7*, 2005. <http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf>.
4. A. van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

5. I. El-Madah and T. Maibaum. Goal-Oriented Requirements Analysis for Process Control Systems Design. In *MEMOCODE 2003*, pages 45–46, France, 2003. IEEE.
6. R. Hassan and S. Bohner and S. El-Kassas and M. Eltoweissy. Goal-Oriented, B-Based Formal Derivation of Security Design Specifications from Security Requirements. In *ARES 2008*, pages 1443–1450, Spain, 2008. IEEE.
7. A. Matoussi and F. Gervais and R. Laleau. A First Attempt to Express KAOS Refinement Patterns with Event B. In *ABZ 2008*, pages 338, 2008. Springer.
8. C. Ponsard and E. Dieul. From Requirements Models to Formal Specifications in B. In *REMO2V'2006*, Luxembourg, June 2006.
9. E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *RE'97*, pages 226-235, 1997. IEEE.
10. N. Lynch and F. Vaandrager. Forward and Backward Simulations - Part I: Untimed Systems. *Information and Computation Journal*, Volume 121, pages 214–233, 1994.